Introduzione al PASCAL

GRUPPO EDITORIALE JACKSON

SYBEX

EDIZIONE ITALIANA

Pierre Le Beux

SOMMARIO

PREFAZIONE

"L'ultima cosa che si trova facendo un lavoro è sapere qual è la cosa che bisogna mettere per prima".

PASCAL,
Pensées

Il linguaggio di programmazione Pascal è stato creato una decina d'anni fa da N. Wirth. L'uso del linguaggio si è sviluppato a partire dal 1970, in primo luogo nelle università e nell'ambiente scientifico. Il suo successo sempre maggiore indica che è destinato senza alcun dubbio a spodestare, negli anni 80, linguaggi come il FORTRAN, i vari derivati dell'ALGOL, il PL/I, etc.

La sua impostazione presenta infatti molti vantaggi rispetto a quella dei suoi predecessori: il Pascal è facile da insegnare e da imparare, non ha le idiosincrasie dei "vecchi" linguaggi, permette di scrivere programmi molto leggibili e strutturati, dispone di nuove facilitazioni per la manipolazione di dati e flussi, generalmente assenti nei linguaggi a vocazione scientifica. In più, e questa è certamente la ragione che gli ha consentito di superare la "barriera" dell'industria, realizzare compilatori (traduttori) del Pascal è estremamente facile. In un'epoca in cui la tecnologia dell'hardware specifico dei calcolatori si sviluppa con particolare rapidità, e con l'arrivo di nuovi microprocessori a 16 bits (il 9900 T.I., l'8086, lo Z8000, l'M68000, il WD9000, etc.), i costruttori devono porsi il problema di scegliere linguaggi evoluti, efficaci, moderni e di facile realizzazione.

Il mercato di questa fascia di costruttori è radicalmente diverso da quello dei costruttori di calcolatori. L'handicap che c'è sulle biblioteche di programmi scritti in linguaggi vecchi (come il FORTRAN, ad esempio) non si pone più negli stessi termini con questi nuovi sistemi. Ad esempio, la facilità con cui si realizzano compilatori Pascal costituisce un criterio economico notevole, che spinge i costruttori a scegliere il Pascal fra tutti i linguaggi evoluti. La strada è stata aperta da alcune case costruttrici di microprocessori come la Texas Instruments e la Motorola. I nuovi utilizzatori, e saranno molti, che "proveranno" il Pascal, non esiteranno a buttare i loro vecchi lin-

guaggi fra i "rifiuti" della storia dei linguaggi informatici. Per coerenza con quanto si è appena detto, in questo libro studieremo il linguaggio Pascal senza far riferimento ad una precedente conoscenza di un altro linguaggio evoluto di programmazione.

Benchè il Pascal sia stato inizialmente sviluppato in inglese, non c'è motivo per non scriverlo "in italiano". Quindi all'inizio del libro gli esempi saranno nelle due lingue, da un lato per facilitarne la comprensione a chi non conosce l'inglese, dall'altro per favorirne lo sviluppo e l'impiego nelle scuole superiori e nei primi anni dell'università.

CAPITOLO 1

INTRODUZIONE

> *"I lumi della geometria, della fisica e della meccanica me ne fornirono il disegno, e mi resero certo che il suo uso sarebbe stato infallibile, se un artefice avesse potuto dar forma allo strumento del quale io avevo concepito il modello..."*
>
> PASCAL,
> Lettera dedicatoria
> della macchina aritmetica (1645)

Il linguaggio Pascal è un linguaggio di programmazione evoluto, detto anche ad alto livello perchè la sua definizione è indipendente dal calcolatore, ovvero dalla macchina, su cui i programmi scritti in questo linguaggio vengono eseguiti. Tuttavia, prima di studiare il linguaggio vero e proprio, è necessario illustrare la struttura di base degli attuali calcolatori, ed i concetti di base della programmazione: queste nozioni saranno utili soprattutto a quanti iniziano ora ad usare un microcalcolatore.

In questo capitolo saranno presentati alcuni concetti di carattere generale sul trattamento dell'informazione, sulla struttura ed il funzionamento dei calcolatori, e infine sugli algoritmi e i metodi di programmazione, tutte cose che non sono proprie del linguaggio Pascal, ma che costituiscono un punto di partenza indispensabile per i principianti e per chi si accinga ad utilizzare questo linguaggio. Pertanto il lettore più esperto può passare direttamente ai successivi capitoli, che trattano il linguaggio vero e proprio.

Nel corso dell'opera esporremo le caratteristiche standard del linguaggio Pascal, ma con un deliberato orientamento verso le realizzazioni interattive, installate su microcalcolatori.

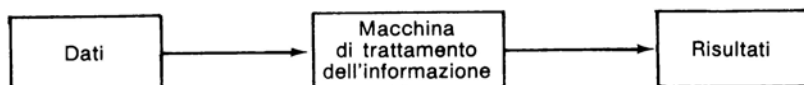
In particolare, faremo spesso riferimento al sistema Pascal sviluppato presso l'U-

niversità di San Diego in California (U.C.S.D.), attualmente disponibile su diversi sistemi a microprocessore.

1 - CONCETTI GENERALI SUL TRATTAMENTO AUTOMATICO DELL'INFORMAZIONE

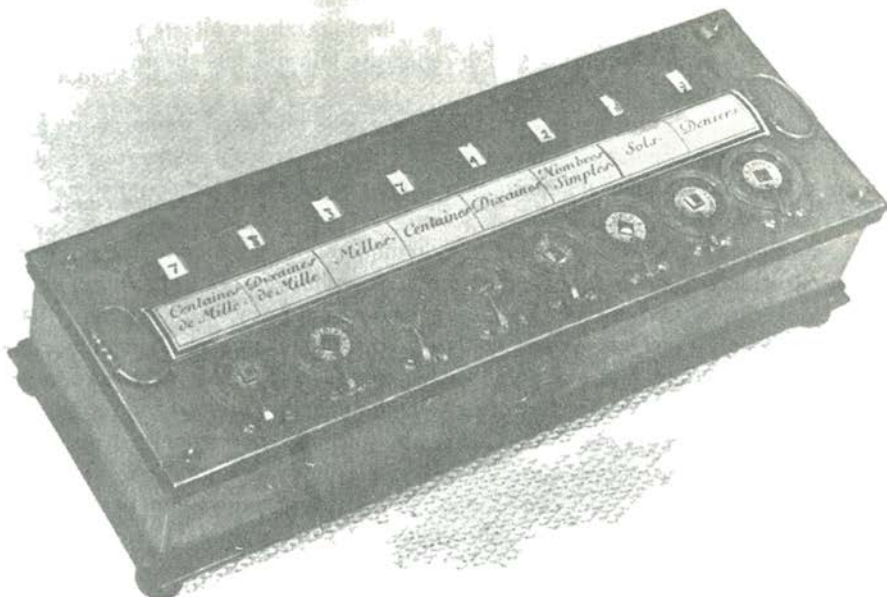
Oggetto e scopo della programmazione è permettere di specificare ad una macchina un determinato lavoro da eseguire in modo automatico.

A tal fine, bisogna fornire alla macchina i valori di determinati parametri, che chiamiamo *dati*. Successivamente, la macchina eseguirà un certo numero di operazioni su questi dati, seguendo un determinato schema di funzionamento che le sarà stato precisato, o una volta per tutte, o su richiesta: questo schema di funzionamento corrisponde a quello che chiamiamo programma. Ultimo passo, tutto ciò si giustifica solo nella misura in cui la macchina fornisce un certo numero di *risultati*.

Si ha dunque schematicamente:

Come si vede nello schema, l'uomo interviene unicamente per alimentare la macchina (dati) e ricevere i risultati. Ovviamente interviene pure nella concezione del programma!

1.1 - Macchine a programma fisso

È importante sottolineare che il trattamento automatico dell'informazione può essere effettuato mediante connessioni meccaniche, elettriche, etc., esistenti fra i differenti elementi che costituiscono la macchina. Parliamo allora di *programma fisso*: nel caso di un sistema elettronico, si dirà cablato. Storicamente, il telaio di Jacquard è stato la prima macchina a programma fisso. Macchine di questo tipo sono necessariamente specializzate in un certo tipo di lavoro automatico: la macchina calcolatrice di Pascal è stata la prima macchina programmata meccanicamente per eseguire operazioni aritmetiche. Fra le macchine a programma cablato si possono citare tutti gli automatismi elettronici, dall'ascensore e dai portelli automatici agli svariati apparecchi automatici usati nell'industria (macchine utensili, catene di montaggio, ecc.).

La caratteristica principale delle macchine a programma fisso è che il programma è codificato in modo definitivo ed irreversibile nella struttura stessa della macchina.

La macchina di Pascal.

1.2 - Macchine a programma registrato

Se si vuole avere una maggiore flessibilità, bisogna ricorrere a macchine *a programma registrato*. Il primo tipo di macchina a programma registrato fu descritto da Babbage (matematico inglese dell'Ottocento), che propose un programma esterno alla macchina, codificato su un supporto continuo. Su questo programma erano dunque registrate le varie istruzioni che costituivano il programma.

Il vantaggio offerto da questo tipo di macchina è che si può cambiare facilmente il programma, e che, di conseguenza, la macchina può trattare diversi tipi di problemi: basta riscrivere il programma. Si dice allora che la macchina è universale, perchè permette di eseguire programmi corrispondenti a tutti i problemi esprimibili sotto forma di algoritmo.

Tuttavia, paradossalmente, macchina universale non è necessariamente sinonimo di complessità, tanto è vero che la macchina di Türing (matematico inglese del Novecento), costituita da una testina di lettura e da un nastro, contenente un programma, capace di muoversi avanti e indietro davanti alla testina di lettura, è in grado di eseguire tutte le operazioni e tutti gli algoritmi eseguibili sulle macchine più moderne.

Il solo inconveniente presentato da questo tipo di macchina è che la programmazione è lunga e noiosa, e inoltre che anche il tempo di esecuzione dei programmi per problemi complessi può essere molto elevato.

È stato necessario il genio del matematico Von Neumann, perchè, in un certo senso, si realizzasse (1946) la sintesi fra le macchine a programma fisso interno e le

macchine a programma registrato esterno. L'idea centrale è stata quella di considerare i programmi come dei dati che si possano immagazzinare all'interno della macchina, in una memoria interna capace di registrare informazioni diverse, cioè sia istruzioni di programma, sia dati.

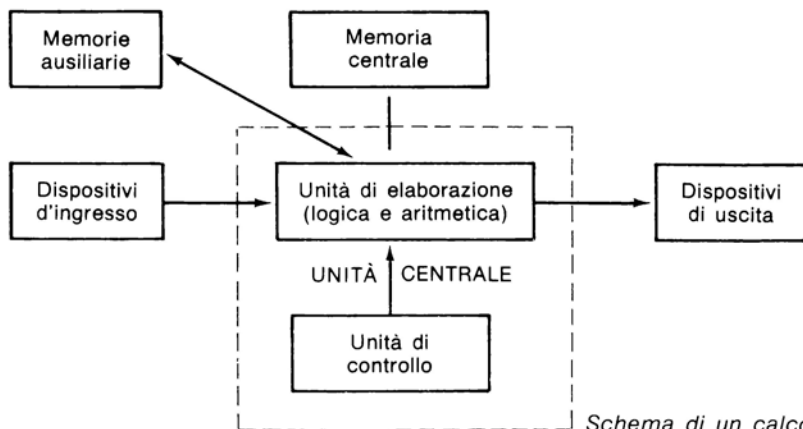
Quest'idea e quest'impostazione sono alla base delle macchine moderne, o calcolatori. Soltanto dopo che i calcolatori sono stati inventati, la programmazione è diventata una disciplina di rilievo, e l'informatica una scienza e una tecnica distinta dalla matematica e dall'elettronica.

Nelle macchine attuali è stato recuperato in modo nuovo il concetto di programma fisso, in particolare quando si parla di *microprogrammazione.* Si tratta infatti di programmi registrati in memorie *di sola lettura*, che non si possono modificare (ROM: Read Only Memory), o si possono modificare solo molto raramente. Queste tecniche permettono di costruire una macchina relativamente complessa, partendo da una "micromacchina" che dispone d'istruzioni elementarissime che servono a scrivere microprogrammi, i quali a loro volta realizzano funzioni e operazioni più elaborate. Comunque noi non ci soffermeremo su questo tipo di programmazione.

Per concludere va solo detto che è possibile registrare i programmi applicativi su memorie di sola lettura di vari tipi: inalterabili (ROM), programmabili (PROM), o riprogrammabili (EPROM).

2 - STRUTTURA E FUNZIONAMENTO DI UN CALCOLATORE

Non è il caso di dare qui una descrizione dettagliata della struttura di un calcolatore, ma è pur sempre necessario conoscere l'attrezzo che si dovrà utilizzare; di qui l'opportunità di illustrare i vari elementi che costituiscono un sistema per il trattamento dell'informazione.

I calcolatori sono detti anche calcolatori numerici, per distinguerli dai calcolatori analogici. Un calcolatore numerico lavora su dati discreti (non continui), codificati per mezzo di numeri binari: il sistema binario è il sistema di numerazione che utilizza la base 2 (v. Appendice 1).

Schema di un calcolatore.

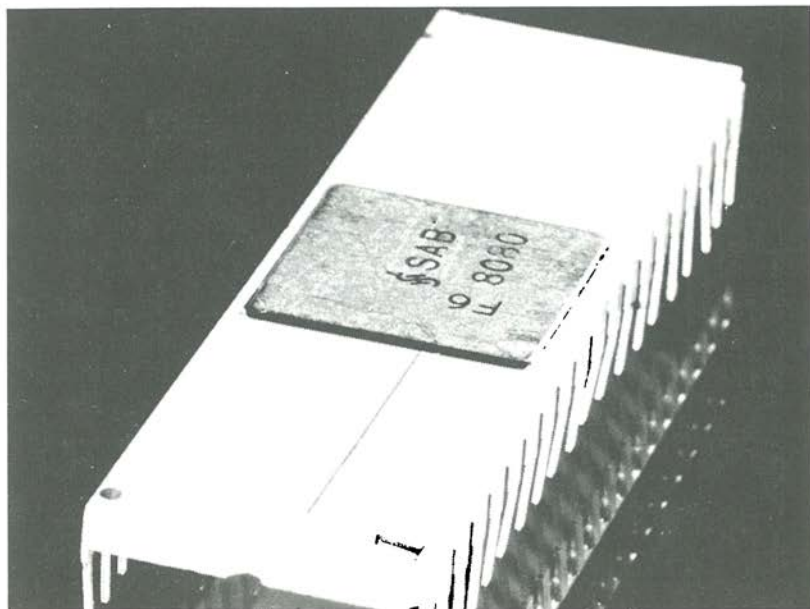
2.1 - L'unità centrale

L'unità centrale è costituita dall'insieme dei circuiti elettronici che eseguono le operazioni richieste al sistema. Queste operazioni sono svolte mediante una serie d'istruzioni che caratterizza il particolare sistema su cui si lavora. Queste istruzioni sono istruzioni elementari aritmetiche, logiche, oppure concernenti lo spostamento dei dati fra la memoria e i dispositivi d'ingresso/uscita.

L'unità centrale può essere scomposta, sia dal punto di vista funzionale che dal punto di vista fisico, in tre unità:

- l'unità aritmetica e logica (UAL);
- i registri (memorie interne all'unità centrale);
- l'unità di controllo, preposta alla decodifica delle istruzioni ed alla loro esecuzione, secondo una sequenza precisa, all'interno dell'unità centrale.

Tutte e tre queste sottounità variano a seconda dei sistemi. L'unità centrale è costruita per lavorare su un quantum d'informazione, detto parola - macchina.

Una *parola-macchina* è caratterizzata dalla sua dimensione, espressa nel numero di cifre, o elementi binari (bits) che la parola medesima può contenere. Così si parlerà di unità centrale ad 8 bits (microprocessore), a 16 bits (minicalcolatore o microprocessore), a 32 bits (mini o grosso calcolatore), etc. La parola-macchina rappresenta altresì il quantum d'informazione che può essere scambiato fra l'unità centrale e la memoria o i dispositivi d'ingresso/uscita. Questi scambi avvengono tramite un in-

Microprocessore.

sieme di fili (bus) che collegano le differenti unità del sistema (bus d'indirizzi, bus di dati, bus di controllo).

La velocità dell'unità centrale, per un'istruzione elementare, è in media dell'ordine del microsecondo.

L'unità centrale è caratterizzata anche dalla sua struttura interna e dalla tecnologia impiegata. Non sempre i sistemi più potenti impiegano le tecnologie più moderne al fine di aumentare la velocità.

Attualmente le unità centrali possono essere integrate su un solo circuito ad alta integrazione (LSI), detto microprocessore.

2.1.1 - L'unità aritmetica e logica

È la parte attiva del sistema: quella che elabora i dati per fornire dei risultati. In questa unità troveremo sempre un'unità di elaborazione aritmetica (addizione, sottrazione, moltiplicazione, divisione, etc.) e logica (confronti e operazioni logiche). Questa unità è associata a delle memorie specializzate, nelle quali vengono effettuate le elaborazioni, e che chiamiamo *registri*: accumulatore, registri indice, pile, (stack in inglese), etc.

L'unità centrale di elaborazione (CPU, dall'inglese Central Processing Unit) è caratterizzata altresì da un repertorio d'istruzioni, che serve ad indicare l'elaborazione da effettuare. Queste istruzioni sono chiamate anche ordini.

Sono istruzioni elementari (sommare due parole, confrontare due parole, etc.), e sono le sole ad essere realmente eseguite dal sistema. Bisogna tenere ben presente che, per l'unità centrale di elaborazione, le istruzioni possono essere assimilate a dei dati, e che tutte le operazioni eseguite dall'unità centrale accedono a parole di memoria.

Attualmente (1979) esiste una *macchina Pascal* realizzata con circuiti a microprocessore (il Western Digital 9000): questo permette di prevedere, per i prossimi anni, un grosso sviluppo di questo linguaggio.

2.1.2 - L'unità di controllo

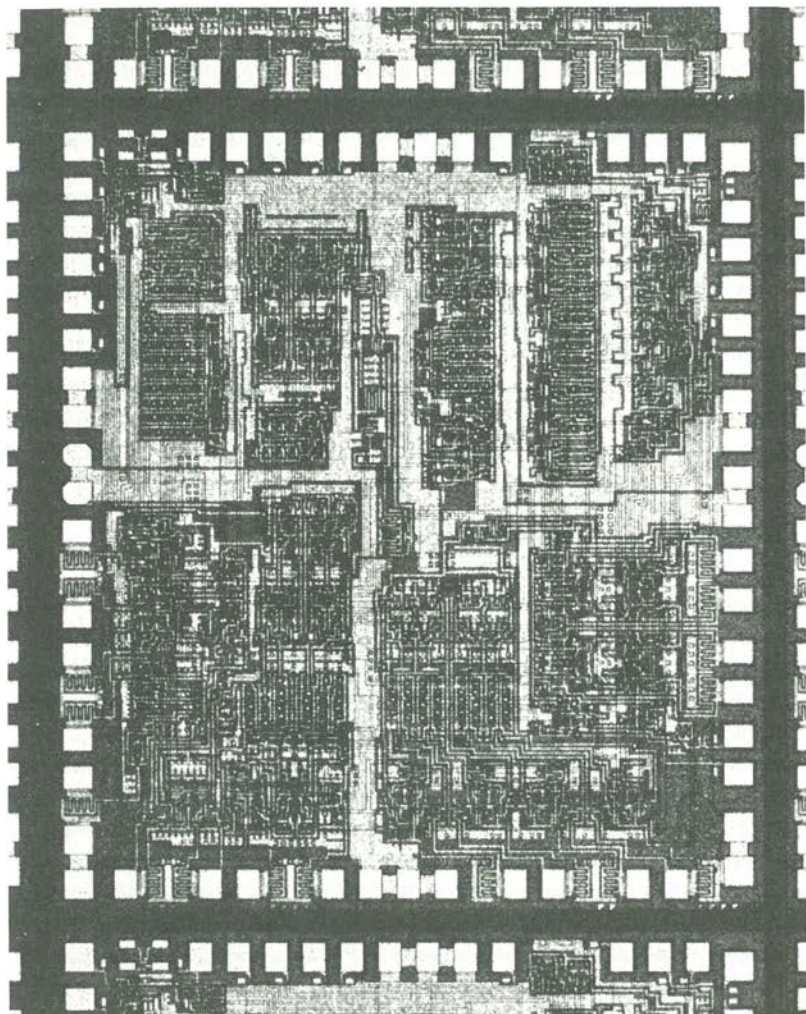
È formata da dispositivi che decodificano le istruzioni e le trasmettono all'unità aritmetica e logica. L'unità di controllo è in relazione con la memoria principale (per leggere o scrivere informazioni in memoria), e con l'unità che svolge le elaborazioni vere e proprie.

I dispositivi di controllo servono anche a decodificare i comandi provenienti da, o diretti all'esterno (dispositivi d'ingresso/uscita).

L'unità di controllo può essere realizzata mediante circuiti elettronici oppure può basarsi su una memoria di sola lettura contenente microistruzioni: in questo caso diciamo che l'unità centrale è microprogrammata.

2.1.3 - I bus

Le informazioni che transitano fra le varie parti del sistema, la memoria e le unità

d'ingresso/uscita viaggiano per mezzo di un insieme di fili elettrici detti *bus*. Si possono distinguere tre tipi di bus: *bus di dati*, su cui viaggiano le istruzioni o i dati provenienti dalla memoria; *bus d'indirizzi*, che specifica le esatte locazioni (indirizzi) di memoria dei dati o delle istruzioni; *bus di controllo*, che contiene i segnali di controllo relativi alle varie parti del sistema. Questa struttura sarà illustrata in seguito. Dal punto di vista del programmatore, ciò implica che qualunque dato, istruzione di programma o risultato deve passare attraverso il bus dati, un elemento alla volta: è quello che J. Backus chiama il "tubo" di Von Neumann, e che costituisce il collo di bottiglia dei calcolatori attuali.

Circuito ad alta integrazione (chip).

Gli attuali microcalcolatori sono basati, per la maggior parte, su un microprocessore (unità centrale) ad 8 bits. Oggi esistono comunque anche dei microprocessori a 16 bits: l'8086 della Intel, il 9900 della Texas Instruments, lo Z8000 della Zilog, il 68000 della Motorola.

Il WD 9000 della Western Digital è un microprocessore a 16 bits concepito per eseguire il codice P fornito dai compilatori Pascal, e offre pertanto prestazioni maggiori nell'esecuzione di programmi in Pascal (v. oltre).

2.2 - La memoria centrale

La memoria centrale è collegata direttamente all'unità centrale e contiene il programma, o i programmi, e i dati sui quali lavoreranno i programmi (che è la caratteristica principale di una macchina di Von Neumann).

In essa al momento dell'esecuzione sono memorizzati i programmi e i dati. Dal punto di vista tecnologico, la memoria principale è caratterizzata dai seguenti elementi:

- la *tecnologia* impiegata. In passato anelli di ferrite; oggi memorie a semiconduttori MOS (da Metal Oxyde Semi-conductor) e bipolari, memorie a bolle magnetiche, etc.;
- il *tempo di accesso*, che indica il tempo necessario alla lettura di un quantum d'informazione in memoria. Nei sistemi attuali, il tempo di accesso è dell'ordine del microsecondo, μs, mentre nei sistemi più veloci è di alcune centinaia di nanosecondi.

2.2.1 - L'organizzazione della memoria

La più piccola informazione memorizzabile o manipolabile prende il nome di *bit* (contrazione di *binary digit*), e rappresenta una cifra binaria (0 o 1): per l'appunto binary digit vuol dire cifra binaria. La capacità di una memoria è definita mediante il numero complessivo di bits, ma, per facilitare il trattamento e l'indirizzamento dei dati in essa contenuti, la si è divisa in celle elementari costituite da più bits, che vengono chiamate *parole di memoria*. A queste si può accedere mediante un indirizzo che rappresenta la posizione della cella stessa in memoria. Per motivi di omogeneità, tutte le celle hanno la stessa dimensione. Pertanto una memoria si caratterizza anche in base alla dimensione delle sue parole di memoria: 8 bits, 32 bits, e così via. La dimensione varia a seconda dei sistemi e dei costruttori, ma nella maggior parte dei casi si hanno parole multiple della parola di 8 bits: questa è chiamata generalmente *byte*. Nei sistemi piccoli (i microcalcolatori) la parola è in genere di 8 bits.

Nei minicalcolatori oggi si usano per lo più parole di 16 bits; nei sistemi più potenti le parole sono lunghe 32 bits o più. Va comunque osservato che i confini fra micro, mini e grossi calcolatori tendono a variare rapidamente.

2.2.2 - Dimensione della memoria centrale

Fissata la dimensione di una parola, un altro parametro che caratterizza la memoria centrale è il numero complessivo di parole contenute in essa. Questa dimensione complessiva è espressa in multipli di K = 1024 parole ($1024 = 2^{10}$). Parleremo dunque di memoria di 4 K parole, di 16 K, di 32 K, etc. Questo parametro è usato anche per indicare lo spazio occupato da un programma. Ad esempio, se un programma occupa 20 K, non sarà possibile eseguirlo su un sistema dotato di una memoria centrale di 16 K.

2.2.3 - Le tecnologie

Esistono numerose tecnologie per le memorie centrali. In passato si utilizzavano memorie a nuclei di ferrite, tuttora presenti peraltro su alcuni calcolatori. Attualmente il loro pregio più notevole è la capacità di conservare il loro contenuto, anche in assenza di corrente; ma hanno d'altra parte molti svantaggi, e cioè costo, ingombro e consumo elevati.

Le memorie attuali si basano tutte sulle tecnologie dei semiconduttori (transistori MOS o transistori bipolari), ed offrono il vantaggio di poter essere integrate su vastissima scala (diverse decine di migliaia di transistori).

Per quanto riguarda le memorie integrate, bisogna distinguere fra memorie di lettura e scrittura, dette RAM (Random Access Memory), e memorie di sola lettura, dette ROM (Read Only Memory). Le RAM contengono dati e/o programmi, le ROM dati e/o programmi inalterabili, come ad esempio codici di caratteri ed il sistema operativo del calcolatore. Attualmente disponiamo di circuiti integrati di memorie RAM di 64 Kbits, ma la capacità d'integrazione è in continuo aumento.

2.3 - I dispositivi d'ingresso/uscita

Questi dispositivi realizzano la comunicazione del sistema con il mondo esterno, e, per prima cosa, servono a fornire al sistema i programmi e i dati, dalla cui esecuzione si avranno in risposta i risultati. In una parola, permettono all'uomo d'impartire degli ordini al sistema, dunque di utilizzarlo.

I dispositivi d'ingresso/uscita sono detti anche *dispositivi periferici* del calcolatore. Citiamo, ad esempio, i lettori di schede o di nastri perforati, le stampanti, le telescriventi, i plotters, le unità di visualizzazione, etc.

In questa categoria si collocano anche le unità di *memoria ausiliaria*, dalle quali si possono leggere o scrivere informazioni che si debbano conservare più a lungo. Infatti la memoria principale di un calcolatore, proprio per il suo carattere universale e limitato, non viene utilizzata per la memorizzazione delle informazioni (programmi o dati) per un tempo superiore a quello richiesto dall'esecuzione di un programma. Si ricorre quindi a memorie dette ausiliarie, nelle quali si potranno memorizzare informazioni in modo permanente, o quasi.

Unità di visualizzazione.

I dispositivi di memoria ausiliaria sono di due tipi:

- memorie *ad accesso sequenziale* (nastri magnetici, schede, nastri perforati, cassette);
- memorie ausiliarie *ad accesso casuale*: dischi magnetici rigidi o flessibili (floppy disk), etc.

Unità a floppy disk.

Ai diversi tipi di memoria ausiliaria sono associate le periferiche corrispondenti (meccanismo di trascinamento per i nastri magnetici o per le cassette, controllori per i dischi magnetici).

OSSERVAZIONI

Un certo numero di programmi di sistema può essere memorizzato, in forma permanente, in una parte della memoria centrale, quella costituita da memorie di sola lettura (ROM).

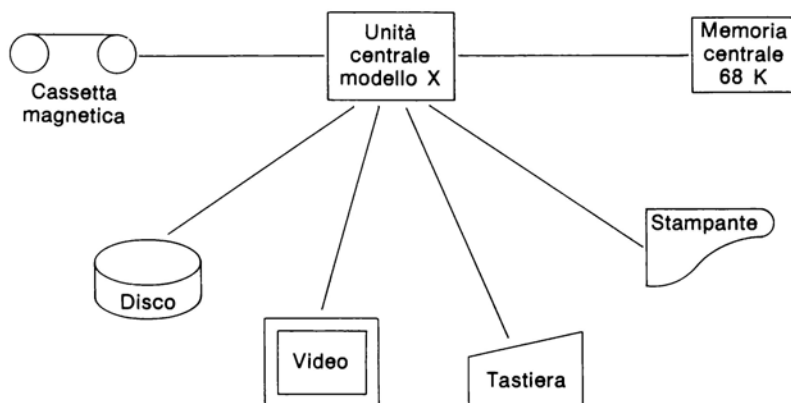
In alcuni casi, conviene considerare le memorie ausiliarie come estensioni della memoria principale. Utilizzando tecniche più sofisticate (overlays, swapping, segmentazione, paginazione), si può dare all'utilizzatore l'impressione di disporre di una memoria quasi illimitata. Queste tecniche, dette di memoria virtuale, non saranno qui illustrate nei dettagli, ma è necessario sapere che esistono, e che si vanno diffondendo sempre più.

Si osservi anche che i dispositivi d'ingresso/uscita sono, in generale, molto più lenti dell'unità centrale (unità di elaborazione + dispositivi di controllo), e che di conseguenza in essi si formano spesso dei colli di bottiglia.

3 - I MICROCALCOLATORI

3.1 - Il concetto di configurazione

Nel considerare un calcolatore, non sempre basta conoscere il tipo di sistema che si usa, ma è altrettanto importante conoscere la configurazione disponibile, cioè la dimensione della memoria, le periferiche e le possibili estensioni.

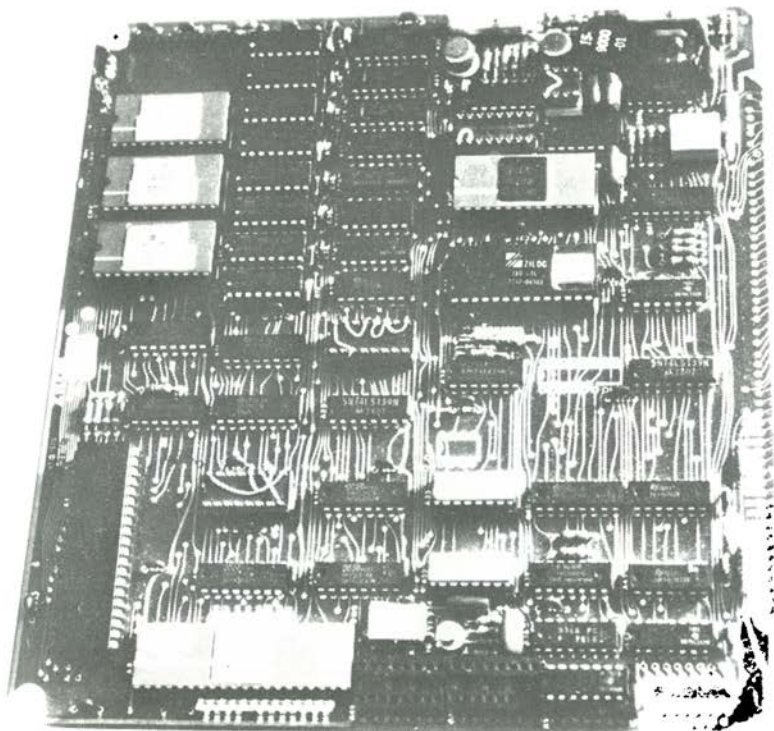
Questa configurazione può essere illustrata con uno schema di questo tipo:

Esempio di configurazione.

Lo schema si riferisce alla configurazione-tipo di un microcalcolatore.
Per calcolatori più potenti, nello schema compariranno un numero ed una varietà molto maggiori di periferiche.

3.2 - Configurazioni dei microcalcolatori

Un microcalcolatore è un calcolatore la cui unità centrale è un microprocessore. Un microprocessore è un circuito ad alta integrazione (LSI), che ha in sè tutte le funzioni di un'unità centrale.
Esistono diverse categorie di microcalcolatori.

3.2.1 - I microcalcolatori su circuito unico

Sono microprocessori che incorporano una memoria di sola lettura (ROM), contenente un programma inalterabile, ed una piccola memoria di lettura e scrittura (RAM). Circuiti di questo tipo sono usati per applicazioni molto semplici, sviluppate in grande quantità.
Attualmente non sono ancora programmabili in un linguaggio evoluto.

Microcalcolatore su singola scheda.

3.2.2 - I microcalcolatori su singola scheda

Si tratta di schede che ospitano un microprocessore, memorie di sola lettura (ROM) contenenti dei programmi, memorie di lettura e scrittura (RAM) destinate a contenere programmi e dati, e circuiti d'interfaccia d'ingresso/uscita (PIO).

Inizialmente, i microcalcolatori su scheda erano programmabili solo in linguaggio macchina; oggi esistono sistemi su singola scheda che permettono di sviluppare programmi in BASIC, mediante la connessione con un'apposita tastiera. Esistono anche microcalcolatori su scheda che permettono di programmare in linguaggio evoluto Pascal (scheda WD 9000); questi utilizzano un microprocessore che è stato sviluppato appositamente per interpretare ed eseguire il codice intermedio del linguaggio Pascal (il codice P).

3.2.3 - I microcalcolatori personali

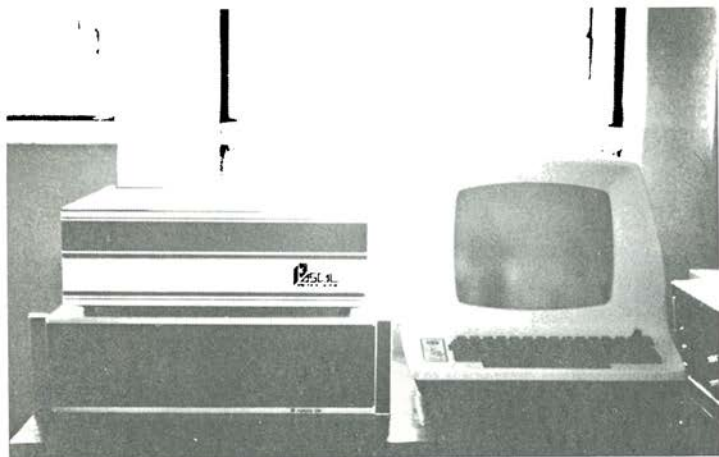
Sono sistemi completi e compatti, basati su un microcalcolatore su scheda, ma che comprendono una tastiera ed un'unità di visualizzazione, con possibilità di connessione a periferiche standard, come unità a floppy disk, piccole stampanti, cassette magnetiche, etc. Un hardware di questo tipo può essere programmato in un linguaggio evoluto, in particolar modo in BASIC ed in Pascal.

Il microcalcolatore personale Apple.

3.2.4 - I microcalcolatori professionali

Questi sistemi, dal punto di vista interno, si basano su schede dello stesso tipo di quelle dei microcalcolatori personali, ma rispetto a questi sono più completi, e possono venir collegati a diversi tipi di periferiche di uso professionale: unità a più dischi, stampanti veloci, nastri magnetici, linee di trasmissione dati, una molteplicità di terminali, etc.

Di fatto, i sistemi di questo tipo possono oggi essere confusi con i minicalcolatori: dispongono di un software sofisticato (multiprogrammazione, gestione di archivi), e possono venir programmati in diversi linguaggi evoluti (BASIC, FORTRAN, COBOL, Pascal, etc.). Va detto che attualmente alcuni sistemi personali possono essere considerati come sistemi professionali.

I sistemi professionali si possono utilizzare in applicazioni commerciali o industriali, come alternativa ai minicalcolatori, e ad un prezzo di acquisto molto più basso.

Sistema Pascal "micro - engine".

L'elemento comune a tutti questi sistemi (tranne i microcalcolatori su circuito unico) è che dispongono tutti, attualmente, del linguaggio BASIC, e che il secondo linguaggio utilizzato è il Pascal: questo senza dubbio soppianterà il FORTRAN presso un gran numero di utilizzatori. Il Pascal è un precursore del linguaggio Ada, che si svilupperà nel corso degli anni '80.

OSSERVAZIONI

A proposito del problema della linea di demarcazione che delimita la categoria dei microcalcolatori, si è visto che una parte di essi (quelli professionali) può essere as-

similata attualmente ai minicalcolatori; si può dire pertanto che questa linea di demarcazione non ha più molto senso, visto che nel 1979 sono apparsi dei microprocessori a 16 bits di potenza uguale a quella dei minicalcolatori di livello medio. Comunque quel che è certo è che la programmazione dei microcalcolatori si fa oggi, e sempre più si farà in futuro, con linguaggi evoluti. Il software è sempre più sofisticato, e, aumentando continuamente il numero degli utilizzatori, si può supporre che lo sviluppo di nuove applicazioni verrà fatto con linguaggi evoluti, arrivando per tale via a disporre di prodotti software di massa, utilizzabili su una grande varietà di sistemi, a costi relativamente bassi in ragione dell'alto numero di esemplari distribuiti. E il Pascal e l'Ada saranno certamente i linguaggi di sviluppo delle applicazioni trasferibili da un sistema all'altro.

È questo uno dei caratteri fondamentali dell'ingresso dei microcalcolatori sul mercato: fra poco il software potrà essere sviluppato su vastissima scala, il che d'altra parte non esclude la possibilità che nasca una nuova categoria d'informatici, "artigiani" e fornitori di prodotti software su ordinazione, ad uso di specifici utilizzatori.

4 - IL SOFTWARE E IL CONCETTO DI ALGORITMO

Il *software* concerne tutti i programmi e tutte le tecniche di programmazione di un calcolatore.

Compito del programmatore è essenzialmente quello di esprimere i problemi da risolvere sotto forma di *algoritmi*, e di tradurli in programmi nel linguaggio stabilito.

4.1 - Note storiche ed etimologiche

Nonostante le apparenze (v. l'algoritmo di Euclide), il termine "algoritmo" non deriva da una parola greca o latina, ma è una contrazione, e deformazione, del nome del matematico arabo Al Khwarismi, che pubblicò due importanti opere: una di aritmetica, ed un'altra il cui titolo *Kitab al-jabr wal muqabala* (tradotto con *L'arte di numerare ed ordinare le parti di un tutto*) è all'origine della parola "algebra".

Quando, tre secoli dopo, il primo di questi due libri fu tradotto in latino, prese il titolo di *Algorismus*. Il termine designava allora il metodo della numerazione posizionale, usato oggi in aritmetica, e la cui scoperta risale in realtà agli indiani, che la trasmisero all'Europa per il tramite degli arabi. Quindi, per la sua origine, il termine "algoritmo" ha qualcosa a che vedere con il più usuale concetto di procedimento di calcolo.

4.2 - Alcune definizioni del termine "algoritmo"

La prima definizione del termine "algoritmo", nella sua accezione attuale, è stata data dal matematico Markov: *"Qualsiasi insieme di regole precise che definisca un procedimento di calcolo destinato ad ottenere un determinato risultato partendo da determinati dati iniziali".*

Gli algoritmi obbediscono alle seguenti tre proprietà (assolutamente necessarie):

a) sono costituiti da un insieme di regole precise, e sono universalmente comprensibili;
b) si applicano a dati che possono variare in larga misura;
c) tendono ad un risultato, che si ottiene quando i dati sono stati scelti correttamente.

Questo concetto di algoritmo può essere assimilato ai più usuali concetti di "metodo", "istruzioni", "tecniche", "procedimento", "regola", etc. Nell'ambito specifico dell'informatica questa definizione non è del tutto sufficiente. Bisognerà quindi aggiungervi alcuni elementi di carattere operativo, quali:

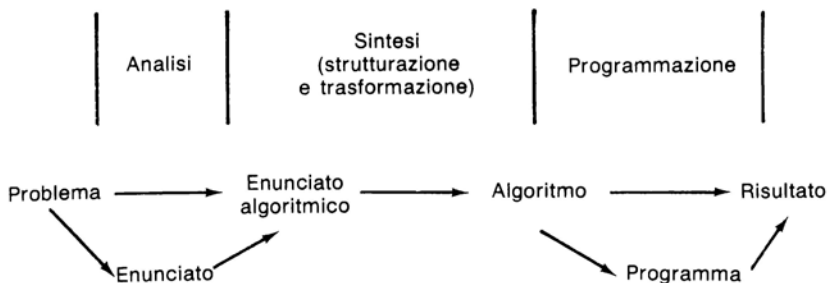
- un algoritmo dev'essere finito: è un insieme finito d'istruzioni, ciascuna delle quali deve a sua volta concludersi entro un tempo finito;
- i processi che intervengono in un algoritmo sono di natura non continua, ma discreta;
- l'azione dell'algoritmo è deterministica: per gli stessi dati deve sempre dare gli stessi risultati;
- esiste un operatore che effettua o esegue le istruzioni (quest'operatore potrà essere una persona, un apparecchio meccanico o elettronico, etc.);
- esiste un supporto di memorizzazione che permette d'immagazzinare non solo i risultati intermedi e finali, ma anche le regole e i dati.

La miglior definizione di algoritmo attualmente valida in campo informatico è stata data da Knuth, nel libro intitolato *The Art of Computer Programming:*

"È un insieme di regole (o istruzioni) aventi le seguenti cinque caratteristiche:

- *dev'essere finito, e concludersi dopo un numero finito di operazioni;*
- *dev'essere definito e preciso: ogni istruzione dev'essere definita in forma non ambigua;*
- *se ci sono dati in ingresso, il campo di applicazione dev'essere precisato (ad esempio, numeri interi, reali, caratteri, etc.);*
- *deve fornire almeno un risultato (dato in uscita);*
- *dev'essere eseguibile: tutte le operazioni devono poter essere eseguite esattamente, e in un tempo finito, da un uomo che utilizzi mezzi manuali".*

4.3 - Problemi, algoritmi e programmazione

Dato un problema concreto, prima di tutto si formulerà un enunciato preciso del problema stesso (ad esempio, calcolare il minimo comun divisore di due numeri), si passerà quindi all'enunciato tramite l'algoritmo, che indicherà le diverse fasi dell'algoritmo, sotto forma di uno schema che sarà poi tradotto in un programma per ottenere i risultati del problema posto.

La sintesi illustrata qui sopra è indispensabile per ottenere dei programmi semplici e ben strutturati. In questa fase si può ricorrere alle tecniche della programmazione strutturata e di trasformazione dei programmi.

È necessario notare che un dato problema non conduce necessariamente ad un unico algoritmo. Diciamo che un problema è *risolvibile* se esiste almeno un algoritmo che permetta di risolverlo. Esistono infatti dei problemi non risolvibili, in quanto non esiste alcun algoritmo che conduca ad una soluzione.

È opportuno sottolineare che il fatto che per un problema esista una soluzione, vuoi per un caso particolare, vuoi per più casi specifici, non vuol dire che il problema sia risolvibile, e che, inversamente, il fatto che un problema non sia risolvibile non vuol dire che non ha soluzioni, ma solo che non esiste nessun metodo che permetta di ottenere tali soluzioni, in tutti i casi in cui tali soluzioni si hanno.

Quindi, come conseguenza di tutto quanto si è detto, un algoritmo può essere considerato come una serie di azioni da compiere allo scopo di risolvere un problema. In seguito daremo alcuni esempi di algoritmi, via via più complessi, dai quali apparirà come essi trovino applicazione in programmazione.

Dobbiamo comunque avvertire il lettore alle prime armi che *non esiste nessun metodo* per trovare, da soli o collettivamente, un algoritmo relativo ad un problema non ancora risolto. La formulazione di un algoritmo è dunque un atto essenzialmente *creativo*, che richiede non solo intelligenza, ma anche intuizione e, in una certa misura, esperienza.

D'altra parte, uno stesso problema può essere risolto con più tipi di algoritmi. Questo è ad esempio il caso dell'ordinamento di una serie di numeri secondo l'ordine crescente o decrescente: numerosi algoritmi danno una soluzione a tale problema, come si vedrà nel Capitolo 4.

In pratica, possiamo dunque distinguere tre tipi di problemi:

- I problemi che siamo in grado di risolvere praticamente, in tutti i casi, usando strumenti manuali o concettuali. L'analisi dei metodi utilizzati per risolvere questi problemi deve portare alla formalizzazione di un algoritmo.
- I problemi per i quali è già stato trovato un algoritmo. Per essi potrà essere interessante ricercare un altro algoritmo, più semplice o che porti più rapidamente al risultato. E questo è un lavoro di ricerca informatica.
- I problemi che non siamo in grado di risolvere in tutti i casi, o non completamente.

L'analisi passa per una fase di ricerca, che può essere senza sbocco. Se il problema è irrisolvibile, non vi è algoritmo; se è risolvibile, bisogna innanzitutto semplificarlo, poi cercare di generalizzarlo. Questa è un'operazione ben più difficile, il cui risultato non è sempre garantito.

4.4 - Esempi di algoritmi

Riportiamo qui a titolo di esempio diverse categorie di algoritmi.

Algoritmo di ricamo

L'impiego di algoritmi è, di fatto, molto esteso, e non si pensi che sia appannaggio esclusivo dei matematici o degl'informatici.

Per convincersene, basta esaminare un algoritmo per la realizzazione di un motivo ricamato su maglia, che troveremo su una qualunque rivista specializzata. Orbene, per il "punto maglia", abbiamo le istruzioni seguenti:

1) Lavorate da destra a sinistra e dall'alto al basso, su file orizzontali.
2) Portate l'ago sotto il lavoro, e puntatelo al centro di una maglia, dal dietro in avanti, sotto la prima maglia da ricamare.
3) Fate uscire l'ago sul dritto del lavoro.
4) Puntate l'ago a destra della maglia che si trova sopra la maglia da ricamare e fatelo uscire a sinistra della stessa maglia.
5) Puntate ancora l'ago, sul dritto del lavoro, al centro della maglia che si trova sotto la prima maglia, passate da destra a sinistra sul rovescio del lavoro.
6) Fate uscire l'ago, puntandolo, come avete fatto prima, al centro della maglia seguente, dal dietro in avanti, sotto la seconda maglia da ricamare.
7) Ripetere le istruzioni da 2 a 6 fino a ricamare tutte le maglie della fila necessarie per formare il motivo.

Quest'esempio c'interessa per più di un motivo: da un lato, include istruzioni dichiarative, come *Lavorate da destra a sinistra*; dall'altro, prevede due distinti tipi d'istruzioni: istruzioni di inizializzazione, che permettono di cominciare il lavoro, e una serie d'istruzioni da ripetersi *finchè* una certa condizione è realizzata.

Pur essendo questo un algoritmo che si riferisce ad un lavoro manuale, risponde tuttavia a tutti i criteri che sono stati definiti per gli algoritmi. La serie d'istruzioni che abbiamo riportato potrebbe costituire benissimo il programma di una macchina per ricamo: basterebbe solo eliminare tutte le parole superflue della lingua parlata, ed avere comandi formulati in una sintassi non ambigua, che definiscano il movimento dell'ago necessario per realizzare il punto.

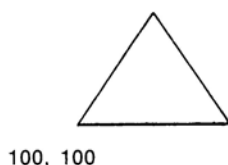
Algoritmo di tracciamento grafico

L'esempio precedente era completamente estraneo al campo informatico, per

quanto possano essere impiegati algoritmi simili per realizzare dei tracciati grafici su un terminale opportuno.

Possiamo avere ad esempio un algoritmo semplice come questo:

- porsi sul punto 100, 100;
- tracciare 100 punti in orizzontale;
- ruotare di 120 gradi;
- tracciare 100 punti;
- ruotare di 120 gradi;
- tracciare 100 punti.

Questo insieme d'istruzioni permette di disegnare un triangolo equilatero:

Un algoritmo analogo, ma più generale, permette di disegnare figure geometriche più complesse; ad esempio, un esagono:

- porsi su un punto di origine;
- *ripetere:*
  - tracciare un lato di lunghezza l;
  - ruotare di 60°;
- *fino a* 6 volte.

Analogamente, per ottenere un poligono regolare di n lati, avremo:

- porsi sul punto di origine;
- leggere n;
- *ripetere:*
  - tracciare un lato di lunghezza l;
  - ruotare di 360°/n;
- *fino a* n volte.

Algoritmo di calcolo semplice

Calcolo del reddito pro capite familiare.

I dati in questo caso sono i seguenti: il numero delle persone *p* adulte della fami-

glia, il numero dei bambini *b* e il reddito *rd* della famiglia. Sapendo che un adulto conta per una parte e un bambino per metà parte, si avrà l'algoritmo seguente:

- siano p, b ed rd i dati
- numero delle parti: np = p+ b/2
- reddito pro capite familiare: q = rd/np

Questo è un algoritmo lineare, perchè abbiamo fatto in modo di lavorare su dati vicini alla soluzione del calcolo.

Algoritmo di matematica elementare

Risoluzione di un'equazione di primo grado:

$$ax + b = 0$$

È un problema elementare che siamo in grado di risolvere algebricamente.

La prima istruzione è una dichiarazione che specifica i dati del problema. La seconda istruzione verifica una condizione su *a* (= 0) che, se trovata valida, implica un altro test su *b*. Infatti, per *a* diverso da 0, sappiamo che il risultato è $x = -b/a$.

Si avrà il seguente algoritmo:

- siano i dati a e b
- se a = 0 *allora* *se* b = 0 *allora* risultato indeterminato
sennò risultato impossibile
- *sennò* risultato x = –b/a.

Algoritmo iterativo

Il calcolo del reddito pro capite familiare diventa più complesso se definiamo una famiglia come un insieme d'individui dei quali conosciamo l'età ed il reddito. In quest'ipotesi, se un individuo ha un'età inferiore a 18 anni, lo considereremo come figlio a carico. Avremo allora l'algoritmo seguente:

- sia n il numero degl'individui
- inizializzare np = 0, rd = 0 e i = 0
- *finquando* i < n *fare:*
  - leggere età(i), r(i)
  - *se* età(i) < 18 *allora* aggiungere 1/2 ad np
sennò aggiungere 1 ad np
  - aggiungere r(i) ad rd
  - aggiungere 1 ad i
- calcolare q = rd/np.

Questo algoritmo parte dal presupposto che un individuo di meno di 18 anni provvisto di reddito possa essere a carico, e che i due coniugi abbiano più di 18 anni. Se invece si vuole tener conto di tutti i casi possibili, bisogna inserire un altro dato, e cioè il grado di parentela: l'algoritmo diventerà in tal modo più complesso. La differenza fra quest'algoritmo ed il precedente sta appunto nel fatto che è il parametro età quello che permette di distinguere i figli dai genitori.

Algoritmo di gioco

Calcolo dei punti di una mano a bridge.

Il concetto di algoritmo è estremamente utile anche nei giochi. Ad esempio, giocando a bridge, per poter dichiarare bisogna contare i punti che si hanno in mano, sapendo che

- l'asso vale 4 punti;
- il re vale 3 punti;
- la dama vale 2 punti;
- il fante vale 1 punto.

Le altre carte hanno valore nullo, ma la mancanza di un colore è valutata 2 punti, e un singleton (una sola carta di un colore) 1 punto. Sappiamo inoltre che una mano è formata da 13 carte.

Nell'algoritmo che segue, chiamiamo *colore(i)* il colore di una carta (fiori, quadri, cuori, picche), e *valore(i)* il valore della carta. Per semplificare l'algoritmo, supponiamo che i valori vadano da 2 a 14. I valori 11, 12, 13, 14 corrispondono rispettivamente a fante, dama, re e asso.

Ncolore rappresenta il numero delle carte di ciascun colore. Abbiamo allora il seguente algoritmo:

- inizializzare punto = 0 ed ncolore (1 ... 4) = 0
- i = 1
- *ripetere*
- *secondo che* colore(i) = fiori, quadri, cuori, picche
    - *fare* g = 1, 2, 3, 4
    - aggiungere 1 ad ncolore(g)
    - sottrarre 10 da valore(i)
    - *se* valore(i) > 0 *allora* aggiungere
        valore(i) a punto
    - aggiungere 1 ad i *finché* i > 13
- *per* g che va da 1 a 4
    - *se* ncolore(g) = 0 *allora* aggiungere 2 a punto
    - *se* ncolore(g) = 1 *allora* aggiungere 1 a punto
- *fine* ciclo.

Algoritmo non numerico

Ricerca di un nome in un elenco.

Quelli presentati negli esempi precedenti erano essenzialmente algoritmi di calcolo. Il problema della ricerca di un nome in un elenco non è risolvibile mediante un calcolo, ma può nondimeno venir definito sotto forma di algoritmo:

– sia nome da cercare
– inizializzare i = 0
– *finquando* nome - elenco(i) non vuoto *fare*
  se nome = nome - elenco(i) *allora* trovato. Stop
  sennò aggiungere 1 ad i.
– indicare "nome inesistente". Stop.

Questo è un algoritmo di ricerca sequenziale, che si arresta o quando si trova il nome cercato, o quando si arriva alla fine dell'elenco. Esistono comunque algoritmi più veloci, soprattutto se l'elenco è in ordine alfabetico.

5 - IL SOFTWARE E LA PROGRAMMAZIONE

Abbiamo visto nei paragrafi precedenti che gli attuali sistemi, o calcolatori, possono eseguire un numero limitato di operazioni elementari, che chiamiamo istruzioni o comandi.

La potenza dei calcolatori sta nel fatto che essi possono eseguire lunghe sequenze d'istruzioni elementari in un tempo brevissimo: ad esempio, un'unità centrale, con un ciclo di base di circa un microsecondo potrà eseguire fino ad 1 milione d'istruzioni al secondo. È evidente che, se si dovessero definire un milione d'istruzioni diverse con le quali tenere occupato il calcolatore per un secondo, il tempo dedicato ad un lavoro tanto noioso non potrebbe essere compensato neppure dai vantaggi che se ne ricaverebbero.

Orbene, la programmazione consiste appunto nell'individuazione di metodi iterativi ricorrenti o ricorsivi che permettano di risolvere determinati problemi, previa formulazione di un numero relativamente limitato d'istruzioni, che dovranno, per contro, essere ripetute dal sistema diverse centinaia, o anche diverse migliaia di volte.

5.1 - Necessità di una codifica e di un linguaggio simbolico

Essendo il linguaggio binario il solo "linguaggio" che il calcolatore conosce, è indispensabile essere in grado di specificare le sequenze d'istruzioni e i dati in una forma alla quale l'uomo possa più facilmente accostarsi.

Si ricorre pertanto a notazioni simboliche, che permettono di scrivere le azioni da compiere in un linguaggio simbolico.

Esistono vari livelli di simbolismo, a seconda che si usi o meno un linguaggio strettamente legato alla macchina.

5.2 - Il concetto d'istruzione simbolica

Per tradurre in programma il problema della soluzione dell'equazione di primo grado $y = ax + b$, potremmo ricorrerere ad un linguaggio simbolico di questo tipo:

Istruzione 1: leggere dalla memoria i valori di a e di b, poi eseguire l'istruzione 2.
Istruzione 2: se il contenuto di a è nullo,
allora se b è nullo scrivere "indeterminato",
sennò scrivere "impossibile",
sennò eseguire l'istruzione 3.
Istruzione 3: calcolare x = −b/a
Istruzione 4: scrivere il messaggio: "la radice è". Scrivere x.

Un linguaggio siffatto ha sì il vantaggio di essere universalmente comprensibile, ma presenta anche diversi inconvenienti:

- per ciascuna istruzione elementare è richiesta una frase relativamente lunga;
- occorre essere in grado di decodificare e tradurre ciascuna frase a beneficio del sistema, in forma non ambigua: un errore di ortografia o l'omissione di un parametro potrà rendere l'istruzione incomprensibile per il sistema;
- per un problema complesso, la sequenza d'istruzioni sarà relativamente lunga; sarà pertanto difficile seguire la concatenazione logica delle operazioni.

Sono possibili diverse semplificazioni: un modo è quello di organizzare le istruzioni in forma sequenziale: nel nostro esempio, si potrà omettere d'indicare, nell'istruzione 1, la necessità di eseguire poi l'istruzione 2. Tale organizzazione sequenziale implicita è messa in opera in *tutti* i linguaggi di programmazione. In altre parole, leggendo o scrivendo un programma dall'alto verso il basso, le istruzioni sono eseguite l'una dopo l'altra, salvo indicazione contraria (interruzione della sequenza).

Ogni istruzione simbolica si compone di tre tipi di parametri: un parametro d'identificazione, un parametro che indica l'operazione o le operazioni da eseguire, ed i parametri operandi.

Tornando all'esempio precedente, esso potrà dunque essere programmato nel modo seguente:

1. leggere i dati a e b
2. *se* a = 0 *allora* *se* b = 0 *allora* scrivere "indeterminato"
   sennò scrivere "impossibile"
   sennò x = −b/a
3. scrivere "la radice è", x

Da questo esempio semplicissimo vediamo che nel formulare un linguaggio di programmazione si può scegliere fra due soluzioni, la prima delle quali consiste nel fare in modo che il linguaggio simbolico sia prossimo alle istruzioni realmente eseguite dal sistema: abbiamo allora un linguaggio *assemblatore*.

Nei cosiddetti linguaggi simbolici assemblatori bisogna precisare ogni operazione

elementare da effettuare; si tratta dunque di una soluzione che ha lo svantaggio di essere strettamente legata al sistema usato, e di richiedere tempi di programmazione abbastanza lunghi. Ad ogni modo, la trattazione di questi linguaggi esula dai limiti di questo libro.

L'altra possibile soluzione consiste nel definire un linguaggio che sia indipendente dal sistema sul quale s'intende eseguire il programma. Ed è a questa soluzione, consistente nell'usare un linguaggio diciamo universale, detto anche linguaggio evoluto o ad alto livello, che ci riferiremo nello svolgimento di questo libro.

Dunque, sempre riferendoci all'esempio precedente, potremo descrivere le operazioni da compiere servendoci delle istruzioni del linguaggio Pascal, che ricalca in larga misura la formulazione algoritmica del problema. Riassumiamo i vantaggi di un linguaggio evoluto, o universale, in confronto ad un linguaggio assemblatore: concisione, possibilità di usare espressioni aderenti alla formulazione matematica, comprensione quasi immediata della natura del programma.

Nondimeno, nei linguaggi di questo tipo è necessario obbedire a precise regole sintattiche, che permettono la traduzione automatica del programma nel linguaggio della macchina. E va detto pure che l'esecuzione del programma sarà generalmente più lenta che con un linguaggio assemblato.

5.3 - I concetti di programma sorgente e di programma oggetto: compilazione e interpretazione

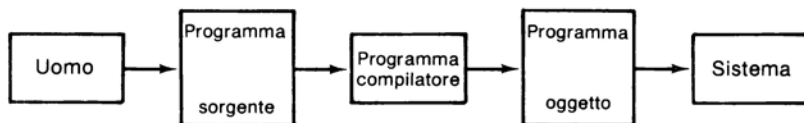
Quando si usa un qualsivoglia linguaggio simbolico (assemblatore o evoluto che sia), bisogna necessariamente tradurre tale linguaggio in una sequenza d'istruzioni-macchina.

Esistono due tipi di traduzione: globale, o *compilazione*, e simultanea, fatta in fase di esecuzione, o *interpretazione*.

5.3.1 - La compilazione

Il linguaggio nel quale è scritto il programma costituisce il *programma sorgente*; la forma nella quale il programma stesso verrà eseguito è detta *programma oggetto*.

L'operazione di traduzione che abbiamo chiamato compilazione è eseguita da un programma che prende il nome di *compilatore*.

Schematicamente, si ha:

Rispetto al programma compilatore, il programma sorgente ha la funzione di dati, il programma oggetto a sua volta ne è il risultato. È il programma oggetto quello che viene eseguito dal sistema.

Anche il programma compilatore viene eseguito dal sistema, ed è generalmente scritto nel linguaggio assemblatore del sistema, per ragioni di efficienza.

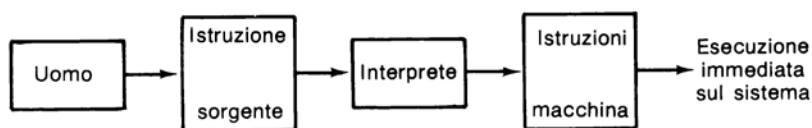
In fase di compilazione, il programma compilatore potrà rilevare e segnalare un certo numero di errori sintattici o semantici, che impediscono di produrre un programma oggetto non ambiguo. Il programma oggetto potrà essere eseguito solo dopo che tutti gli errori siano stati eliminati.

Comunque è importante sottolineare che un programma compilato senza errori non è necessariamente un programma privo di errori; potrà infatti contenere errori logici, che non vengono rilevati a livello di compilazione.

Come ultima osservazione, nel caso di un linguaggio universale il programma sorgente può essere tradotto, e in seguito eseguito, su qualsiasi sistema dotato di un compilatore corrispondente al linguaggio sorgente: questo presuppone la standardizzazione dei linguaggi di questo tipo. Per contro, l'esecuzione del programma oggetto sarà possibile solo a patto che i sistemi siano affini, o compatibili, ed abbiano la medesima configurazione.

Nella pratica, tuttavia, accade abbastanza spesso che più compilatori dello stesso linguaggio differiscano leggermente da una versione all'altra, il che porta come conseguenza la non trasferibilità dei programmi sorgente. Il problema può essere risolto o usando gli standard meno elastici, o modificando un certo numero d'istruzioni, quando si debba passare da un sistema ad un altro. Il linguaggio Pascal non è completamente standardizzato, ma le differenze fra le diverse realizzazioni sono minime.

5.3.2 - L'interpretazione

Secondo quest'altra tecnica di traduzione del linguaggio evoluto, il linguaggio sorgente viene tradotto, istruzione per istruzione, al momento della sua esecuzione. L'operazione è effettuata da un *interprete*, che traduce ciascuna istruzione in linguaggio macchina, e la esegue immediatamente.

Schema di un interprete.

Qui non esiste un programma oggetto, e le istruzioni vengono tradotte ogni volta che il controllo è dato ad una di esse. L'unico riferimento è dunque il programma sorgente.

Un linguaggio interpretato è generalmente *interattivo* (v. i linguaggi BASIC e APL). Questa proprietà permette di sviluppare e verificare i programmi con grande rapidità, senza dover passare attraverso le fasi di compilazione, collegamento dei moduli, caricamento, esecuzione, ecc.

Il linguaggio Pascal è caratterizzato dal fatto di utilizzare un codice intermedio (il *P-code*) che può venire interpretato; comunque, nel suo complesso, possiamo considerarlo come un linguaggio compilato.

5.3.3 - Vantaggi e svantaggi dei linguaggi compilati e dei linguaggi interpretati

I linguaggi compilati hanno il vantaggio di essere meglio strutturati e, in genere, più leggibili. Inoltre vengono tradotti una volta sola, e perciò forniscono un programma oggetto che non dovrà più essere tradotto ad ogni esecuzione. Il tempo globale di esecuzione è molto inferiore a quello di un programma interpretato.

I linguaggi interpretati, dal canto loro, permettono di sviluppare e modificare un programma in maniera interattiva, cosa che facilita la messa a punto dei programmi. Come contropartita, l'esecuzione è più lenta, e in alcuni casi si hanno programmi mal strutturati, difficili da capire o da modificare.

Bisogna comunque segnalare l'esistenza di compilatori Pascal non interpretati, che permettono di sopperire agl'inconvenienti di un linguaggio interpretato. Il vantaggio che deriva dall'utilizzo di un interprete del codice P è che dà la possibilità di sviluppare molto rapidamente dei compilatori su nuovi sistemi. È questa la ragione per la quale le case costruttrici di microcalcolatori hanno optato per questo linguaggio.

6 - I SISTEMI OPERATIVI DEI CALCOLATORI

Abbiamo visto che l'uso dei linguaggi simbolici richiede che ci sia almeno un programma di base: il compilatore o l'interprete. In realtà, la programmazione fatta in un linguaggio evoluto implica l'esistenza di altri programmi, detti di utilità, e la necessità di disporre di biblioteche di sottroprogrammi: citiamo, in particolare, tutti i programmi che eseguono le operazioni d'ingresso/uscita, le funzioni matematiche quali radici quadrate, logaritmi, esponenziali, etc.

Bisogna poi disporre anche di un insieme di programmi che costituisce il *sistema operativo*, il cui compito è quello di gestire l'insieme delle periferiche e la sequenza delle operazioni da eseguire: lettura del programma sorgente, compilazione o interpretazione, edizione di un listato, visualizzazione dei risultati, esecuzione del programma oggetto, edizione dei risultati. Tutte queste operazioni vengono realizzate sotto il controllo di un programma detto *di sistema*. In alcuni casi, si parla di *programma supervisore*, ma il termine è in genere riservato ai sistemi che lavorano in tempo reale, vale a dire che ciascun dato dev'essere elaborato immediatamente, o in un intervallo brevissimo.

Esistono poi dei programmi di utilità (*editors*), che permettono di riprendere un programma per correggerlo; e, ancora, dei programmi (*debuggers*) di aiuto nella messa a punto dei programmi sviluppati. Infine, se si hanno a disposizione delle memorie ausiliarie, ci saranno anche programmi di gestione degli archivi su disco (*file system*).

6.1 - Monoprogrammazione e multiprogrammazione

Questi due termini stanno ad indicare il tipo di sistema utilizzato, e non un metodo o un tipo di programmazione.

Un sistema lavora in monoprogrammazione se, in un dato momento, un solo programma è in elaborazione da parte del calcolatore: in altre parole, se un solo programma è presente nella memoria centrale. In questo caso, data la grande differenza di velocità fra le periferiche e l'unità centrale, si ha come conseguenza che l'unità centrale può essere inattiva per un certo tempo. Pertanto, quando si hanno molte operazioni d'ingresso/uscita, si ha un rendimento di utilizzo molto basso, il che è tuttavia di scarsa incidenza se si lavora su un sistema interattivo, in cui il tempo di risposta dev'essere rapportato ai tempi dell'operatore. I microcalcolatori dispongono in genere di software per elaborazioni in monoprogrammazione, perchè vengono usati da un solo utilizzatore per volta.

I sistemi che lavorano in multiprogrammazione utilizzano il tempo d'inattività corrispondente alle operazioni d'ingresso/uscita di un programma per eseguire un altro, o diversi altri programmi. In questo modo aumenta notevolmente il rendimento di utilizzo dell'unità centrale, ma nello stesso tempo aumenta anche la complessità del sistema. Troviamo software di questo tipo nei microcalcolatori più sofisticati.

Fra i sistemi a multiprogrammazione si collocano anche i sistemi a divisione di tempo (*time - sharing*), che consentono un dialogo permanente con più terminali (stampanti o video): in questo caso gli utilizzatori vengono serviti alternativamente per brevissimi intervalli di tempo, in modo che tutti possano procedere contemporaneamente nei loro lavori. Questo sistema viene usato sui calcolatori più potenti.

6.2 - Linguaggi di comando e di controllo

È altresì necessario, sulla base della maggiore o minore complessità dei sistemi operativi, dare al sistema delle istruzioni che gli indichino il tipo di lavoro da svolgere.

Ogni sistema necessita di un linguaggio tramite il quale sia possibile specificare che si desidera compilare un programma, o avere un listato, o ancora eseguire un programma.

Sui grossi sistemi non interattivi, questo avviene mediante un certo numero di comandi espressi sotto forma di *schede di controllo*, che devono precedere e seguire il programma. Uno dei principali inconvenienti di questi linguaggi di controllo è che non sono standardizzati, e variano da un calcolatore all'altro; per di più, hanno spesso una sintassi inutilmente astrusa e rigida.

I microcalcolatori, invece, offrono, fra gli altri, il vantaggio di utilizzare linguaggi di comando estremamente semplici, e anzi d'immediata comprensione per l'utilizzatore. È evidente che la realizzazione dei programmi su un sistema di questo tipo ne è facilitata.

Citeremo più avanti, a titolo di esempio, i principali comandi utilizzabili sul sistema Pascal dell'Università della California di San Diego (U.C.S.D.), oggi disponibile su vari microcalcolatori.

CAPITOLO 2

ALGORITMI E CONCETTI GENERALI SULLA PROGRAMMAZIONE

"Ora, di questi due metodi, l'uno del persuadere, l'altro del dilettare, darò qui soltanto alcune regole del primo; e sempre che si sia convenuto sui principi e ci si mantenga fermi nel riconoscerli: altrimenti non so se possa esistere un'arte capace di adattare le prove all'incostanza del nostro capriccio".

PASCAL,
De l'esprit géométrique

In questo capitolo presenteremo in sintesi le caratteristiche del linguaggio Pascal, facendo riferimento fondamentalmente al rapporto sul linguaggio Pascal di N. Wirth (1973): questo rapporto descrive lo standard attuale del linguaggio. Nel documento l'Autore sottolinea che il linguaggio è stato sviluppato come attrezzo pedagogico per l'insegnamento della programmazione, e che alla prima formulazione del 1970 sono state apportate delle modifiche tendenti a garantire una maggior attitudine del linguaggio ad essere trasferito su calcolatori diversi.

Prima di entrare nel merito delle principali caratteristiche del linguaggio, torneremo sul concetto di algoritmo, e sui suoi differenti modi di rappresentazione. Gli esempi saranno volutamente elementari, per permettere ai principianti di familiarizzarsi subito con le strutture di base del linguaggio. Nella prima parte del capitolo presenteremo alcuni modi di rappresentazione degli algoritmi.

Una precisazione: il concetto di diagramma di flusso sarà introdotto solo per ragioni di documentazione, in quanto sarà poi abbandonato nel resto dell'opera. Infatti u-

seremo sempre e soltanto rappresentazioni sotto forma di alberi algoritmici o di grafi GNS (che saranno definiti in seguito).

Sempre in questo capitolo, introdurremo anche il concetto di diagramma di struttura.

1 - SUGLI ALGORITMI ELEMENTARI

Abbiamo visto che, prima di passare alla programmazione di un problema, occorre analizzarlo in modo da formulare, o trovare un algoritmo che permetta di risolverlo: questo procedimento dovrà necessariamente precedere qualunque attività di programmazione.

Un problema è caratterizzato innanzitutto da un enunciato che, in una prima fase, può essere scritto, o verbalizzato, in un linguaggio prossimo alla lingua parlata. Ma compito primario dell'informatico, e di chiunque intenda studiare un problema allo scopo di tradurlo in programma, è quello di formalizzarlo in modo preciso. Questa fase passa per l'identificazione dei *dati* del problema e dei *risultati* da ottenere, il che impone, in particolare, di definire il tipo ed il campo di variazione dei dati e dei risultati, esattamente come si fa nella matematica elementare.

Siamo poi abituati ad iniziare l'enunciato di un problema con: *siano x ed r...*; dunque anche un programma inizierà con dichiarazioni dello stesso tipo.

La seconda fase consiste nel definire il metodo da usare per ottenere i risultati cercati, a partire dai dati: è senz'altro la fase più difficoltosa, ed è quello che chiamiamo algoritmo.

A questo livello si presentano diversi problemi. Prima di tutto, il programma dovrà essere espresso in una forma calcolabile, e questo vale anche per i problemi non numerici! In un primo tempo, poco importa se la forma adottata non è la più semplice o la più efficace.

Attualmente esistono metodi di prove formali a cui si può ricorrere per dimostrare la correttezza di un algoritmo. Esistono pure tecniche di trasformazione dei programmi che permettono di ottenere versioni semplificate, o più efficaci; ma questo esula dal tema del libro, che è l'iniziazione al linguaggio Pascal. Ai lettori che fossero interessati all'argomento consigliamo la lettura di: J. Arsac, *La construction de programmes structurés*.

Qui ci limiteremo ad illustrare le tecniche e gli strumenti della programmazione strutturata. L'espressione, spesso usata un po' a sproposito, non rimanda necessariamente ad un metodo universale di formulazione di un programma, e neppure di un algoritmo, a partire dall'enunciato di un problema; tutt'al più tali tecniche permettono di definire un quadro, uno schema ed una struttura che servono ad orientare il pensiero nella direzione della soluzione del problema proposto. A questo scopo si potrà ricorrere innanzitutto al metodo detto degli *affinamenti successivi*, introdotto da N. Wirth: il metodo consiste nel cominciare con il semplificare il problema in modo grossolano, e poi aggiungere, una fase dopo l'altra, dei particolari che permettano di pervenire alla soluzione.

Il metodo può essere anche visto come un'analisi dall'alto verso il basso, a cui fa riscontro inizialmente un primo livello di scomposizione del problema.

Ogni sottoproblema viene poi scomposto in elementi più dettagliati, fino ad arrivare al livello più elementare, che corrisponde ad un'istruzione dell'algoritmo.

Vedremo che una procedura di questo tipo può essere rappresentata mediante strutture ad albero o strutture a scatole cinesi, donde il nome di programmazione strutturata.

Il linguaggio Pascal si adatta in modo particolare a questo tipo di analisi, e si vedrà come le diverse strutture che via via incontreremo corrispondono ad istruzioni disponibili nel linguaggio di programmazione.

Per cominciare, studieremo i differenti livelli di complessità degli algoritmi, cominciando da un esempio assai semplice; in un secondo tempo introdurremo direttamente i corrispondenti programmi in Pascal.

1.1 - I differenti livelli di complessità degli algoritmi

1.1.1 - La formula

Si abbia il seguente problema: calcolare la retribuzione lorda di una persona, conoscendo la retribuzione oraria ed il numero di ore lavorative.

Si può arrivare alla soluzione mediante il seguente enunciato algoritmico:

sia h la retribuzione oraria
sia t il numero delle ore
allora la retribuzione lorda $l = h \times t$

L'algoritmo corrispondente è immediato, e si può riassumere come segue:

dati h, t reali
$l = h \times t$
risultato l

In questo caso, l'algoritmo si riduce ad una sola formula di calcolo.

La verifica è ugualmente immediata.

1.1.2 - Sequenza di formule con risultati intermedi

Possiamo rendere leggermente più complesso il problema precedente, e formularlo come segue:

Calcolare la retribuzione netta di una persona, conoscendo la retribuzione oraria e il numero di ore lavorative, e considerando le trattenute per oneri sociali.

In tal caso nell'enunciato algoritmico precedente dovremo aggiungere:

sia p la percentuale delle trattenute per oneri sociali sulla retribuzione di base
allora, le trattenute sono: $s = l \times p$,
e la retribuzione netta: $n = l - s$

O anche, in una forma più condensata:

dati h, t, p reali
$l = h \times t$
$s = l \times p$
$n = l - s$
risultato n

Qui l'algoritmo è costituito da una sequenza di più formule, con risultati intermedi *l* ed *s*.

Si osservi che l'ordine delle operazioni corrispondenti alle formule non è indifferente: *l* dev'essere calcolata prima di *s*, ed *s* prima di *n*.

Avremmo potuto ricorrere alla proprietà di sostituzione di una formula con un'altra: in tal modo il risultato *n* potrebbe venir calcolato in una sola formula:

$$n = h \times t\ (1 - p)$$

Vedremo che questa proprietà non è sempre utilizzabile in programmazione, soprattutto in riferimento all'istruzione di assegnazione, definita più avanti.

1.1.3 - Gli algoritmi condizionali e la struttura di selezione

Rendendo l'esempio precedente ancora più complesso, si può fissare un tetto per le trattenute per oneri sociali, oltre il quale non si abbiano ulteriori trattenute.

L'enunciato algoritmico dovrà allora essere integrato come segue:

dati h, t, p, max
$l = h \times t$
$s = l \times p$
se s > max *allora* s = max
$n = l - s$
risultato n

Abbiamo introdotto qui un test su una condizione conseguente ad un calcolo: è quello che chiamiamo struttura di selezione semplice, perchè la condizione verificata può essere vera o falsa. Per ciascun termine dell'alternativa si preciserà l'elaborazione da effettuare.

1.1.4 - Gli algoritmi iterativi

Gli algoritmi che abbiamo appena visto sono validi per una sola serie di dati. Se vogliamo invece calcolare la retribuzione di più individui ricorrendo agli algoritmi precedenti, dovremo inserire un procedimento iterativo che agisca sull'insieme delle persone: in tal modo l'algoritmo diventerà iterativo. Assumendo *i* come indice sequen-

ziale delle persone, e supponendo di fermarsi quando si sia trovato il dato $h(i) = 0$, avremo:

```
dati comuni p, max
i = 1
ripetere
        leggere dati h(i), t(i)
        l(i) = h(i) × t(i)
        s(i) = l(i) × p
        se s(i) > max, allora s(i) = max
        n(i) = l(i) − s(i)
        risultato n(i)
        aggiungere 1 ad i
finchè h(i) = 0
```

Quest'algoritmo può essere scritto anche utilizzando un'altra struttura, con la quale si assume che il numero degl'individui (*nind*) sia noto.

```
dati p, max, nind
i = 1
finquando i ≤ nind iterare
      leggere dati h(i), t(i)
      l(i) = h(i) × t(i)
      s(i) = l(i) × p
      se s(i) > max allora s(i) = max
      n(i) = l(i) − s(i)
      risultato n(i)
      aggiungere 1 ad i
fine iterazione
```

E si potrebbe infine usare anche una struttura iterativa siffatta:

```
dati p, max, nind
per i che varia da 1 a nind iterare
      dati h(i), t(i)
      l(i) = h(i) × t(i)
      s(i) = l(i) × p
      se s(i) > max allora s(i) = max
      n(i) = l(i) − s(i)
      risultato n(i)
fine iterazione
```

Si è visto dunque che esistono diversi modi per rappresentare un'iterazione. Li ritroveremo nel linguaggio Pascal.

Questi esempi ci hanno permesso di passare in rassegna l'insieme delle strutture algoritmiche che si possono incontrare in tutti i linguaggi di programmazione. Per completezza bisogna considerare anche gli algoritmi ricorsivi, che esaminiamo qui di seguito.

1.1.5 - Gli algoritmi ricorsivi

Fra le strutture algoritmiche che abbiamo illustrato, distingueremo le strutture di selezione (*test*) e le strutture iterative (*cicli*).

La maggior parte dei linguaggi di programmazione non prevede altre strutture: pertanto solo con queste si possono formulare tutti gli algoritmi.

Vedremo, in particolare, che la struttura a ciclo permette di realizzare programmi ricorrenti.

Esiste nondimeno un altro modo per formulare gli algoritmi, a volte più conciso, e più appropriato alla definizione dei calcoli o delle elaborazioni su strutture di dati: ci riferiamo alla formulazione di algoritmi *ricorsivi*. Il concetto di ricorsività è a volte considerato troppo complesso per i principanti, ma tuttavia è utilissimo in certi casi e, nella misura in cui il Pascal lo permette, non c'è nessuna ragione per non utilizzarlo: è infatti un attrezzo potente, che è necessario conoscere bene. Per ora ci limiteremo a darne una definizione semplice, accompagnata da esempi elementari.

Il concetto di ricorsività è analogo alla definizione di talune funzioni matematiche, nelle quali la funzione serve a definire se stessa.

Cosi ad esempio la funzione fattoriale può essere definita da

$$n! = (n-1)! \times n \qquad \text{con } 0! = 1$$

oppure da

$$\text{fatt}(n) = \text{fatt}(n-1) \times n \qquad \text{con fatt}(0) = 1$$

Analogamente, la somma dei primi n numeri interi può essere definita da

$$\text{som}(n) = \text{som}(n-1) + n \qquad \text{con som}(0) = 0$$

E ancora, il termine di una progressione geometrica può essere definito da

$$p(n) = p(n-1) \times q \qquad \text{con } p(0) = 1$$

Questo modo di definizione è estremamente conciso.

Il calcolo richiede invece che la formula ricorsiva sia applicata finchè non si pervenga ad un valore noto della funzione. Ad esempio,

$$\begin{aligned} fatt(3) &= fatt(2) \times 3 \\ &= fatt(1) \times 2 \times 3 \\ &= fatt(0) \times 1 \times 2 \times 3 \end{aligned}$$

Si ritrova sì la definizione di fattoriale di 3, ma il calcolo non è eseguito. Bisogna riprendere le operazioni nell'altro senso.

Sapendo che *fatt(0) = 1*, avremo:

$$\begin{aligned} fatt(1) &= 1 \times fatt(0) = 1 \\ fatt(2) &= fatt(1) \times 2 = 2 \\ fatt(3) &= fatt(2) \times 3 = 6 \end{aligned}$$

Per calcolare una funzione ricorsiva bisogna dunque dapprima sviluppare la formula ricorsiva, e poi, una volta che si conoscono i differenti elementi della formula, ricalcolarli nell'altro senso.

La formulazione di un algoritmo ricorsivo non è dunque sempre il modo più rapido per giungere ad un risultato: negli esempi che precedono, sarebbero più efficaci degli algoritmi ricorrenti. Per il fattoriale, ad esempio, calcoleremo direttamente prima *fatt(2)*, poi *fatt(3)* moltiplicando *fatt(2)* per 3, e così via, fino a calcolare *fatt(n)* moltiplicando *fatt(n—1)* per *n*.

Ci sono comunque dei casi in cui l'applicazione della ricorsività permette di avere programmi più semplici. Ne vedremo qualche esempio nel prossimo capitolo.

Per il momento, dopo aver introdotto il concetto di ricorsività, vi torneremo solo dopo aver illustrato i concetti di procedura e di funzione.

2 - RAPPRESENTAZIONE DEGLI ALGORITMI

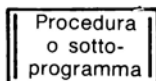
Esistono diversi metodi grafici per la rappresentazione degli algoritmi. Essi permettono di descrivere le varie operazioni, mediante uno schema che indica gli ordini e le condizioni che intervengono nell'algoritmo.

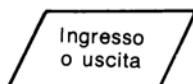
2.1 - I diagrammi di flusso

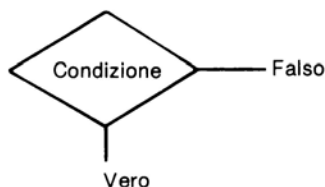
Il metodo più vecchio, e che sempre più cade in disuso, è noto sotto il nome di *diagramma di flusso*. I simboli usati sono:

Il rettangolo

Elaborazione
o calcolo

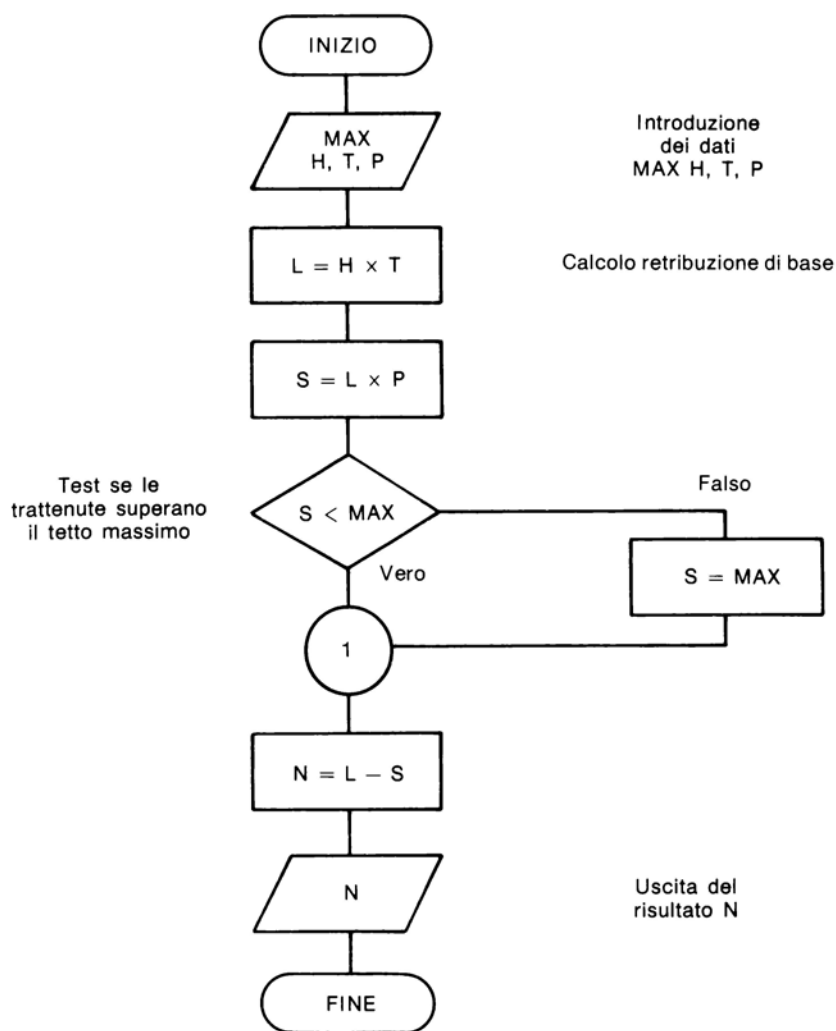
Rappresenta un'elaborazione, o un calcolo, per la quale si hanno una sola entrata ed una sola uscita.

Ancora un rettangolo, ma con due linee verticali, si usa per rappresentare una procedura, o un sottoprogramma. Questo simbolo permette di rappresentare un'elaborazione complessa.

Il parallelogramma

Designa le istruzioni d'ingresso o di uscita. Alcuni tipi di dispositivi d'ingresso/uscita possono venir specificati da simboli speciali di cui parleremo più avanti.

Il rombo

Rappresenta un test, o una selezione. La condizione è espressa all'interno del rombo. Si ha un solo ingresso, e l'accertamento della condizione rimanda a due possibili uscite, l'una corrispondente alla condizione vera, l'altra alla condizione falsa.

I circoletti e gli ovali

Permettono di rappresentare i rimandi e le etichette nel corpo del diagramma di flusso. Nell'esempio, i due simboli rappresentano rispettivamente un'etichetta e un rimando.

Per l'algoritmo condizionale visto nel paragrafo 1.1.3 avremo ad esempio:

2.1.1 - Limiti dei diagrammi di flusso

In questo libro non useremo i diagrammi di flusso come metodo di rappresentazione perchè i diagrammi di flusso fanno nascere delle cattive abitudini nel modo di programmare. Comunque ne parliamo perchè, nell'industria, anche oggi la fase di anali-

si di un problema si conclude spesso con il disegno di un diagramma di flusso il quale, in qualche modo, costituisce la "mappa" dell'algoritmo, che potrà essere poi programmato da un'altra persona, e da un'altra équipe. E di fatto questo schema risponde all'esigenza di una certa divisione del lavoro in un contesto produttivo.

Il diagramma di flusso funge in questo caso da documento base per l'analisi organica, che costituisce il legame fra l'analista ed il programmatore; serve anche a documentare il lavoro dell'analista, ed è, dal punto di vista teorico, indipendente da questo o quel linguaggio di programmazione: un medesimo diagramma di flusso può venir tradotto in programma con più di un linguaggio.

Ma negli ultimi anni i metodi hanno subito un'evoluzione, per cui il diagramma di flusso non è più un passaggio obbligato all'interno di un progetto informatico.

Infatti, per problemi complessi, il diagramma di flusso non entra tutto in una sola pagina, per cui è molto difficile seguirlo e modificarlo. Inoltre, può dimostrarsi errato quando è già cominciata la fase di programmazione, donde correzioni e aggiustamenti su parti già fatte, che si traducono in programmi mal strutturati. Per tutte queste ragioni, alle quali va aggiunta l'esistenza di tecniche di trasformazione dei programmi e di programmazione strutturata, il diagramma di flusso ha perduto quel posto che era suo una decina d'anni fa.

Il linguaggio Pascal permette di fare a meno di questa rappresentazione, ricorrendo ad una maggior varietà di strutture, mal riportabili ad un diagramma di flusso: per questo consigliamo ai lettori che altrimenti resterebbero fedeli a questo tipo di rappresentazione di approfittare dell'occasione offerta loro dall'apprendimento di un linguaggio nuovo per abbandonarlo. I principianti, poi, potranno farne tranquillamente a meno.

2.2 - Altre rappresentazioni schematiche degli algoritmi

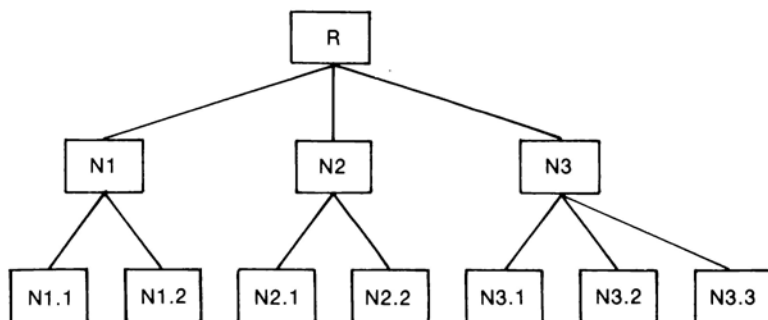
2.2.1 - La rappresentazione ad albero

Gli algoritmi possono venir rappresentati anche in altri modi, in primo luogo mediante *alberi algoritmici.*

Un albero è caratterizzato da un insieme di nodi e di rami, con in più un nodo particolare e unico, detto radice.

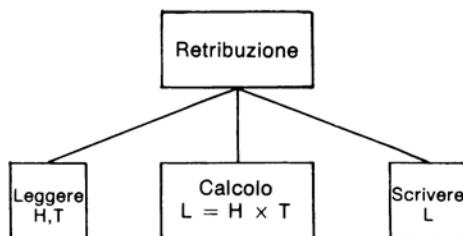
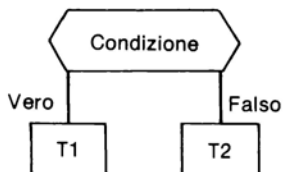
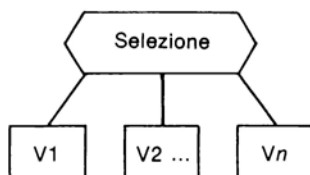
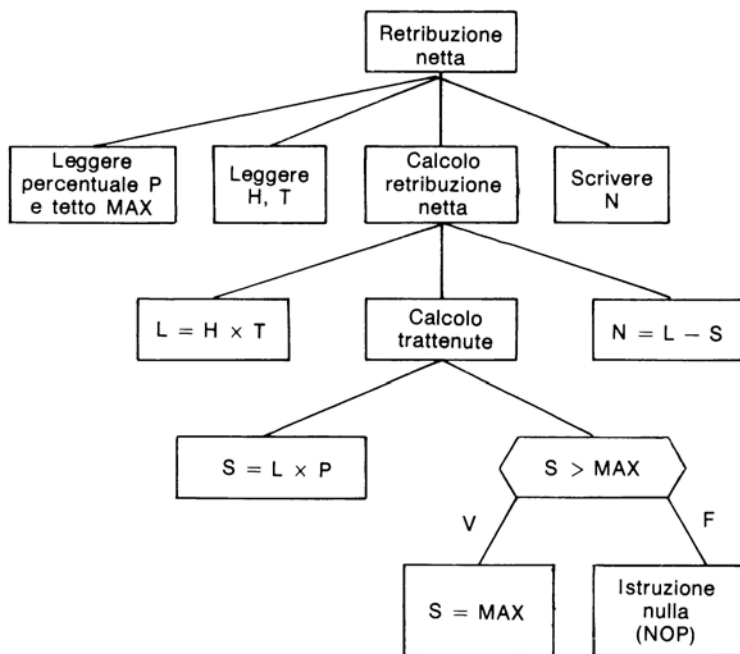
Dal punto di vista formale, un albero può venir definito in modo ricorsivo, come un insieme di nodi *N*, e con un nodo *R* particolare, detto radice; gli altri nodi costituiscono una suddivisione dell'insieme *N-R* in *n* insiemi (*A1, ... An*), che sono a loro volta degli alberi: chiameremo questi ultimi sottoalberi di *R*.

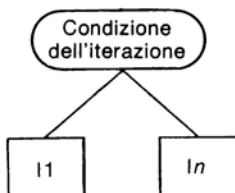
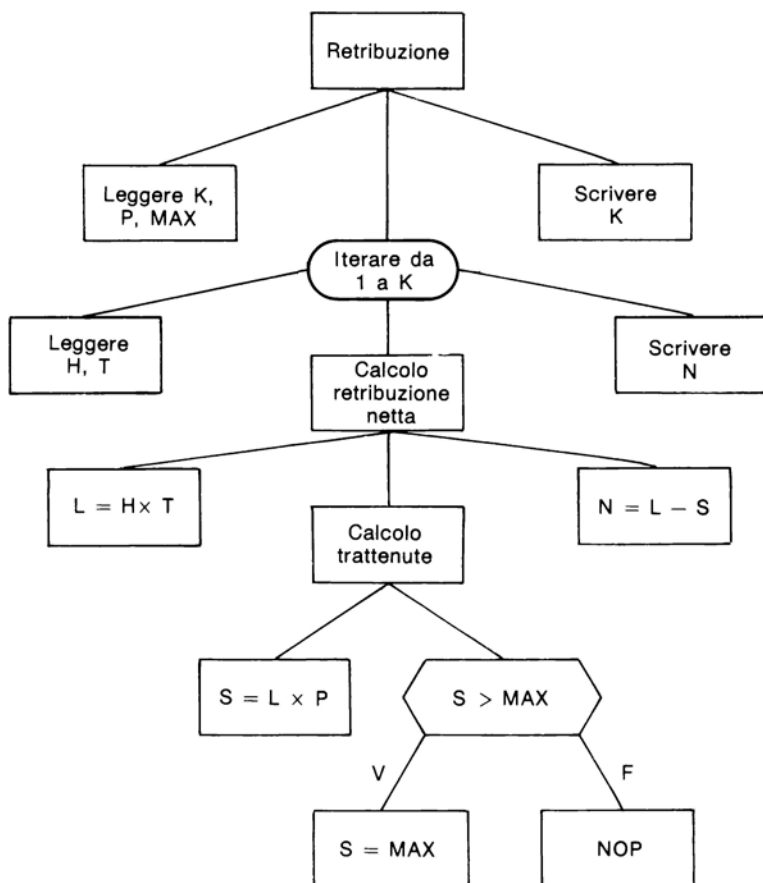
Quando un insieme *Ai* è ridotto ad un unico nodo, diciamo che quello è un nodo terminale.

Schematicamente un albero viene rappresentato con una figura di questo tipo:

I nodi terminali si chiamano foglie. L'albero traduce la gerarchia delle operazioni da svolgere.

2.2.2 - L'albero algoritmico

Le operazioni di un algoritmo vengono rappresentate da una struttura ad albero nel modo seguente: quando un'operazione è scomponibile in operazioni più semplici, queste ultime saranno specificate al livello sottostante. Su uno stesso livello l'ordine delle operazioni è da sinistra verso destra.

Tornando all'esempio del calcolo della retribuzione netta, si ha:

Volendo introdurre le trattenute, e insieme la verifica sul tetto massimo delle trattenute, dobbiamo far intervenire un test, o selezione. Una *selezione semplice* è rappresentata dal simbolismo:

È possibile anche rappresentare *selezioni multiple*, ove ogni ramo della selezione rappresenterà una delle alternative attivabili dalla condizione. Avremo allora:

Nel caso dell'esempio precedente, si avrà:

Resta da rappresentare il concetto *d'iterazione*, che comporta nello stesso tempo un meccanismo di ripetizione ed una condizione che pone termine alla ripetizione.

Nella struttura ad albero, la rappresentazione è la seguente:

La condizione per la continuazione o meno dell'iterazione può essere espressa o mediante una condizione del tipo: *finquando* una condizione è vera; o mediante un comando di ripetizione *finchè* una condizione non diventa vera; o ancora mediante un ordine d'iterazione che farà variare un parametro di controllo da un limite iniziale ad un limite finale. Si abbia, ad esempio, l'algoritmo già visto nel paragrafo 1.1.4, che opera un'iterazione su k individui.

Vediamo che questo modo di rappresentazione è più strutturato di quello costituito dai diagrammi di flusso. Altrettanto semplice, permette d'altra parte di realizzare un metodo di programmazione dall'alto verso il basso: mediante degli affinamenti successivi che facilitano la scrittura di un programma strutturato, si arriva alle operazioni elementari costituenti l'algoritmo.

2.3 - Un altro modo di rappresentazione strutturata degli algoritmi

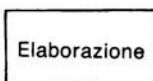
I diagrammi di flusso permettono di dare una rappresentazione grafica, o schematica, degli algoritmi, ma hanno l'inconveniente di permettere rimandi (frecce) ad un punto qualunque della struttura rappresentata. In più, impediscono di rappresentare in modo chiaro il concetto di ciclo iterativo. Possono inoltre dar luogo a programmi mal strutturati, con istruzioni di salto inutili, o addirittura dannose: ad esempio il famoso *goto*, considerato pericoloso da E. Dijkstra, uno dei primi autori che hanno sottolineato il bisogno, e la necessità, di una programmazione strutturata. Appare chiaro che, da quando sono state introdotte le tecniche di programmazione strutturata, l'utilizzazione dei diagrammi di flusso va regredendo, e dovrebbe regredire ulteriormente, tranne che nel caso della rappresentazione di algoritmi puramente lineari, o che presentino poche rotture della sequenza.

Comunque, l'uso di rappresentazioni grafiche non ha perso i suoi caratteri di utilità: ad esempio l'utilità di disporre di una rappresentazione indipendente da questo o quel linguaggio di programmazione, che permetterà di dare un'idea più globale della struttura dell'algoritmo ed, eventualmente, di fornire una spiegazione più immediata, proprio perchè grafica.

Una rappresentazione siffatta, attualmente ancora poco usata, è quella resa possibile dal modello proposto da Nassi e Schneidermann: si tratta dei grafi detti NS, o GNS. Il modello ha il vantaggio di essere facilmente utilizzabile e facilmente comprensibile, e indubbiamente utile dal punto di vista pedagogico, come abbiamo potuto verificare con dei programmatori principianti. Ma non basta: ha anche il vantaggio di permettere la rappresentazione schematica del metodo della programmazione strutturata per affinamenti successivi proposto da N. Wirth, l'inventore del linguaggio Pascal.

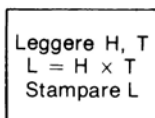
Si tratta di grafi disposti l'uno dentro l'altro, che permettono di rappresentare le elaborazioni sequenziali, i test e le varie strutture dei procedimenti iterativi.

2.3.1 - Il simbolo di blocco di elaborazione

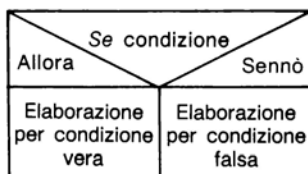
Come nei diagrammi di flusso, il simbolo usato è il rettangolo:

Elaborazione

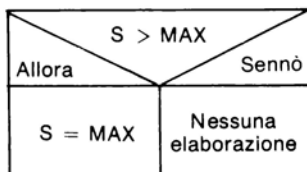
In esso può comparire tanto un'elaborazione elementare (calcolo o operazione d'ingresso/uscita) quanto una sequenza di elaborazioni elementari, senza iterazione nè test.

Considerando l'esempio, già presentato, del calcolo della retribuzione base, avremmo:

Leggere H, T L = H × T Stampare L

2.3.2 - Il simbolo di blocco condizionale, o selezione

È quello che chiamiamo test, e che nei diagrammi di flusso si rappresenta con il rombo. Nelle rappresentazioni GNS è un rettangolo, diviso in tre parti da due segmenti disposti a V. Nella parte superiore si rappresenta la condizione; le altre parti sono simmetriche, e rappresentano le elaborazioni da eseguire nei due casi: a sinistra l'elaborazione che viene eseguita se la condizione è vera, a destra l'elaborazione che viene eseguita se la condizione è falsa.

Otterremo allora lo schema seguente:

Esempi

1) Se, nell'esempio del calcolo delle trattenute sociali, vogliamo verificare se il tetto massimo viene superato, abbiamo:

Qui all'alternativa "falso" non corrisponde alcuna elaborazione.

2) Volendo invece verificare le condizioni per l'esistenza di una soluzione dell'equazione di primo grado $ax + b = 0$, avremo:

Esistono due alternative, la seconda delle quali ($a = 0$) può a sua volta essere scomposta in due nuove alternative, a seconda che b sia o no nullo. Questo simbolismo permette dunque di rappresentare strutture di condizioni (test) annidate. Questa rappresentazione è, a nostro avviso, più chiara di quella ad alberi algoritmici, perchè permette di simulare il processo della programmazione per affinamenti successivi.

Chiaramente, questo procedimento grafico ha i suoi limiti, in rapporto alla possibilità di rappresentare le concatenazioni a blocchi uno nell'altro: per ovviare all'inconveniente possiamo fermarci ad un certo livello, e riprendere la scomposizione un po' più avanti. Nell'esempio precedente, avremmo potuto scrivere:

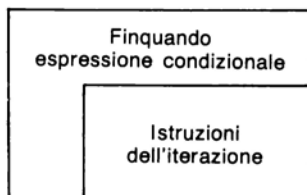
Il caso "soluzione degenerata" può venir scomposto in seguito.

2.3.3 - I blocchi iterativi

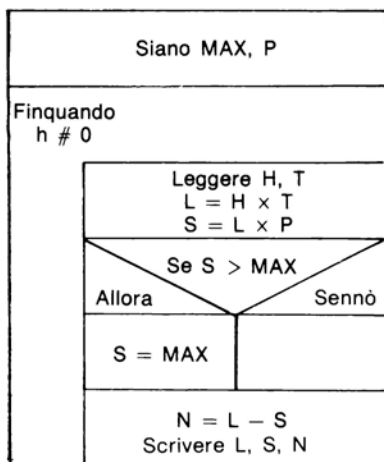
Nel simbolismo dei diagrammi di flusso non esiste, per le iterazioni, altra rappresentazione oltre a quella che utilizza un test ed una freccia che rimanda all'inizio del ciclo.

Ora, si è visto che le strutture iterative possono essere varie: abbiamo a disposizione sia la struttura *finquando*, sia la struttura *per*, sia, infine, la struttura *ripetere...finchè*.

2.3.3.1 - La struttura finquando

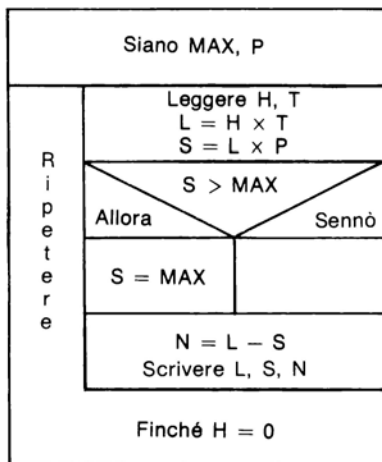
Nel simbolismo GNS la struttura iterativa *finquando* è schematizzata nel modo seguente:

Questa struttura sta a significare che, *finquando* l'espressione sotto condizione è vera, le istruzioni dell'iterazione devono essere eseguite.

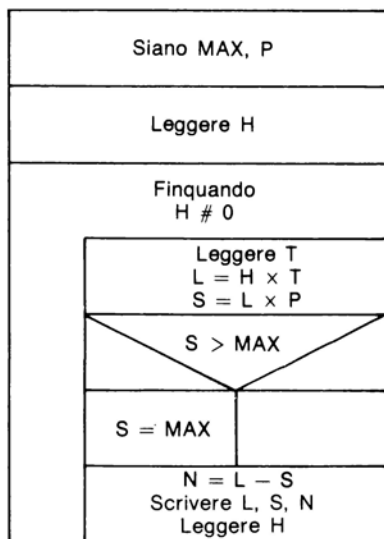
Esempio

Nel caso del calcolo delle retribuzioni di un insieme di persone, il momento di fine iterazione può venir definito in base al fatto d'incontrare, ad esempio, un valore particolare, indicante che non ci sono più dati da elaborare. Supponendo che questo dato sia rappresentato dal valore $h = 0$, otterremo allora la seguente rappresentazione dell'algoritmo.

Nella struttura rappresentata qui sopra, l'elaborazione è effettuata anche per $h = 0$. Ma su questo punto ritorneremo in seguito.

2.3.3.2 - La struttura ripetere

Il simbolismo che si usa per la struttura *ripetere... finchè* è analogo, ma inverso:

Riprendendo l'esempio come è stato definito in 2.3.3.1, otterremo:

Anche qui l'elaborazione su $h = 0$ avviene nell'ultima iterazione: pertanto le due forme iterative *finquando* e *ripetere* sono equivalenti.

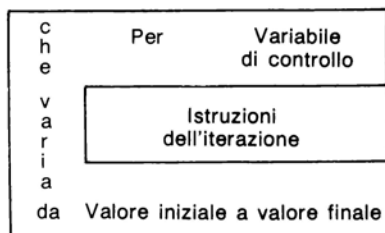
Si può tuttavia modificare la struttura dell'algoritmo *finquando* in modo da far sì che non venga fatta l'elaborazione su $h = 0$; l'algoritmo precedente può essere infatti riscritto in questo modo:

La lettura di un valore nullo di *h* conduce all'arresto dell'iterazione.

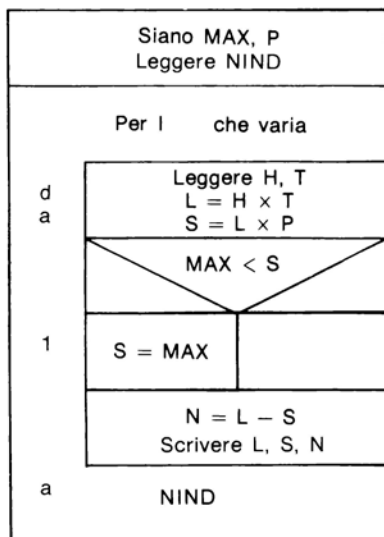
Viceversa non è possibile modificare allo stesso modo la struttura *ripetere*, perchè l'iterazione dev'essere eseguita almeno una volta. Torneremo su questo punto quando parleremo delle corrispondenti istruzioni del linguaggio Pascal.

2.3.3.3 - L'iterazione per

Esiste infine un'ultima struttura iterativa, utilizzabile quando il numero d'iterazioni da eseguire è conosciuto a priori.

Questa struttura può venir rappresentata con il seguente simbolismo:

che varia da | Per Variabile di controllo
Istruzioni dell'iterazione
Valore iniziale a valore finale

L'esempio precedente è rappresentabile con una struttura di questo tipo a patto che il numero delle persone sia noto prima che l'iterazione abbia inizio. Questo nu-

mero dovrà pertanto venir considerato come un dato del problema: *nind*. Avremo allora:

Questa struttura permette di rappresentare un meccanismo iterativo controllato da una variabile di controllo (nell'esempio, *i*) che varia da un valore iniziale (1) ad un valore finale (*nind*, sempre nell'esempio).

Vedremo che questa struttura può essere realizzata anche con le strutture *finquando* e *ripetere*.

3 - LE PRINCIPALI CARATTERISTICHE DEL LINGUAGGIO PASCAL

Un programma è nello stesso tempo una serie di frasi scritte in un linguaggio definito da un alfabeto ed un insieme di regole di formazione delle frasi in quel linguaggio. Le regole costituiscono la sintassi, o grammatica del linguaggio.

3.1 - Il Pascal come linguaggio universale

Il linguaggio Pascal deriva dal linguaggio di programmazione ALGOL, in particolare dall'ALGOL W. Per questo può essere considerato come un linguaggio scientifico, ma d'altra parte dispone d'istruzioni di definizione del tipo carattere e delle strutture di flusso che l'ALGOL non ha, e che lo rendono meglio adatto alle applicazioni gestionali. È vero che l'ALGOL 68 prevede queste possibilità, ma è più difficile da imparare e da usare.

Si può dire dunque che il linguaggio Pascal si adatta a tutti i tipi di applicazione: problemi scientifici, gestionali, di elaborazione di testi, didattica, etc.

In più, contrariamente a quanto accade per molti linguaggi di programmazione, la realizzazione di compilatori Pascal è semplice e rapida. Il passaggio attraverso un codice intermedio, detto *codice P*, permette di definire in tempi brevissimi interpreti del codice P funzionanti su sistemi diversi. Quest'operazione è stata realizzata segnatamente, per vari microcalcolatori, dal gruppo operante presso l'Universitá della California di San Diego (U.C.S.D.).

Da ultimo, le istruzioni d'ingresso e di uscita sono anch'esse molto semplificate, e più flessibili che in molti linguaggi di programmazione meno recenti.

3.1.1 - Il concetto di programmazione in Pascal

Un *programma* si compone di una serie d'istruzioni, assimilabili alle frasi del linguaggio.

Dal punto di vista formale, studiare il linguaggio vuol dire imparare le regole di formazione di queste istruzioni. Dal punto di vista pratico e pedagogico, tale studio non può fare a meno della presentazione di un certo numero di esempi per ciascuna struttura del linguaggio.

In Pascal, un programma si compone di due parti ben distinte:

- una parte d'*intestazione*, che contiene il nome del programma ed i suoi eventuali parametri;
- una parte costituente il *corpo* del programma, che contiene le istruzioni vere e proprie. Questa parte è detta anche *blocco*. Un blocco può essere scomposto in sottoblocchi.

Un programma termina con il carattere punto (.).

Un'*istruzione* in Pascal non ha un formato fisso, come in linguaggi tipo FORTRAN, COBOL e BASIC: non esistono quindi zone predeterminate, nè il formato "scheda".

In Pascal occorre fare una distinzione fra il concetto d'*istruzione semplice* e quella d'*istruzione strutturata*. Il carattere punto e virgola (;) si usa come *separatore* delle istruzioni, e non è quindi un carattere terminatore, come ad esempio nel linguaggio PL/I.

3.1.2 - Esempio di programma

Si abbia il seguente programma in italiano:

```
PROGRAMMA Pascal;
VAR ind, i: intero;
INIZIO
PER ind: = 1 A 20 FARE
  INIZIO
    PER i: = 1 A ind FAR scrivere ('');
    scrivereln ('Pascal è vivo')
  FINE;
FINE.
```

In inglese avremo:

```
PROGRAM Pascal;
VAR ind, i: integer;
BEGIN
FOR ind: = 1 TO 20 DO
  BEGIN
    FOR i: = 1 TO ind DO write ('');
    writeln ('Pascal è vivo')
  END;
END.
```

Nella versione italiana, il programma è sufficientemente chiaro: vengono definite due variabili intere, denominate *i* e *ind*, e poi ha luogo un ciclo iterativo che fa scrivere per venti volte il messaggio 'Pascal è vivo', ogni volta con un margine costituito da un numero di spazi bianchi uguale al numero della riga scritta. Avremo cioè:

```
Pascal è vivo                 prima riga
  Pascal è vivo               seconda riga
    Pascal è vivo             terza riga
```

3.2 - Il Pascal come linguaggio strutturato

Nel paragrafo precedente si è detto che il corpo del programma è chiamato anche blocco. Un blocco è strutturato *a sezioni*, che possono essere costituite o da *dichiarazioni*, cioè istruzioni che non determinano delle azioni durante l'esecuzione del programma, o da *istruzioni eseguibili*, cioè istruzioni che determinano un'azione, o un'elaborazione, da parte del sistema.

Nel corpo di un programma possono non comparire sezioni di dichiarazioni, mentre ci sarà sempre la sezione contenente istruzioni eseguibili.

Nell'esempio precedente, l'istruzione VAR *ind, i: intero; (integer)* è una sezione di dichiarazione che definisce le variabili *ind* ed *i* come numeri interi.

La parte di programma che va da INIZIO (BEGIN) a FINE (END) è la sezione che contiene le istruzioni eseguibili.

3.2.1 - Le diverse sezioni di un programma in Pascal

Il corpo di un programma in Pascal è costituito da un massimo di sei sezioni, le prime cinque delle quali sono facoltative. Le sezioni di dichiarazioni devono precedere, in un certo ordine, la sezione delle istruzioni eseguibili.

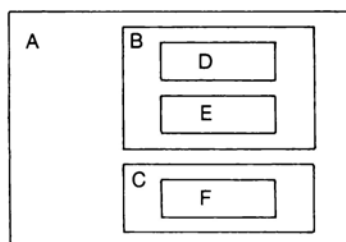
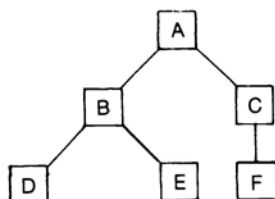
L'ordine delle sei sezioni è il seguente:

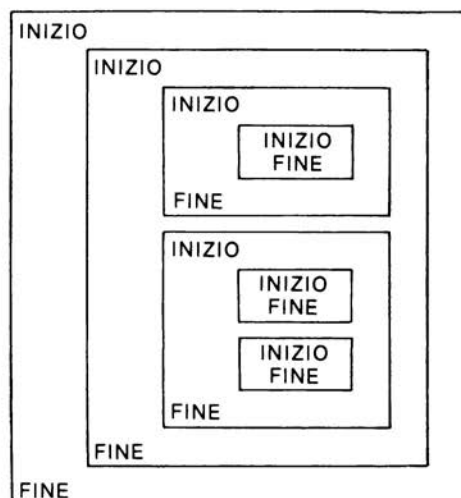
– sezione di dichiarazione delle *etichette*;
– sezione di dichiarazione o di definizione delle *costanti*;

- sezione di dichiarazione o di definizione dei *tipi*;
- sezione di dichiarazione delle *variabili*;
- sezione di dichiarazione delle *procedure* e delle *funzioni*;
- sezione delle *istruzioni eseguibili*.

La struttura di queste istruzioni non sarà esaminata nei dettagli nel presente capitolo, perchè se ne parlerà via via in tutto il corso del libro.

È comunque opportuno osservare subito che le sezioni procedure e funzioni contengono a loro volta dei blocchi, ciascuno dei quali può prevedere tutte le sezioni elencate prima. Diciamo in questo caso che i blocchi sono *annidati*, cioè inseriti gli uni negli altri.

L'annidamento può essere rappresentato da uno schema di questo tipo:

o da una struttura ad albero, che mostrerà la gerarchia di annidamento:

Il blocco A contiene i blocchi B e C (B e C sono annidati in A); il blocco B contiene i blocchi D ed E (annidati in B e, di conseguenza, in A); il blocco C contiene il blocco F (annidato in C ed in A).

Nella sezione delle istruzioni eseguibili, anche le sezioni, o gruppi d'istruzioni, che vanno da INIZIO (BEGIN) a FINE (END) possono essere annidate. Si possono infatti benissimo avere strutture di questo tipo:

Al fine di evidenziare queste strutture all'interno del programma, ricorreremo ad un meccanismo che consiste nello scalare i margini in modo da indicare i livelli di annidamento. La struttura schematizzata sopra si scriverà ad esempio in questo modo:

```
INIZIO
  ..........
  INIZIO
    ..........
      INIZIO
        ..........
      FINE
  FINE
    ..........
  INIZIO
    ..........
      INIZIO
        ..........
      FINE
        ..........
      INIZIO
        ..........
      FINE

  FINE
    ..........
FINE
```

3.2.2 - Esempio di programma strutturato

```
PROGRAMMA media;
VAR
  dato, somma, media: reale;
  i, n: intero;
INIZIO
  scrivereln ('introdurre il numero dei dati');
  leggere (n);
  SE n > 0 ALLORA
    INIZIO somma: = 0;
    scrivereln ('introdurre', n, 'dati');
    PER i: = 1 A n FARE
      INIZIO
        leggere (dato);
        somma: = somma + dato;
      FINE;
    media: = somma/n;
    scrivereln ('media =', media)
    FINE
  SENNÒ scrivereln ('nessun dato');
FINE.
```

La versione inglese del programma è la seguente:

```
PROGRAM media;
VAR
  dato, somma, media: real;
  i, n: integer;
BEGIN
  writeln ('introdurre il numero dei dati');
  read (n);
  IF n > 0 THEN
    BEGIN somma: = 0;
    writeln ('introdurre', n,'dati');
    FOR i: = 1 TO n DO
      BEGIN
        read (dato);
        somma: = somma + dato;
      END;
    media: = somma/n;
    writeln ('media =', media)
    END
  ELSE writeln ('nessun dato');
END.
```

Per ora non disponiamo di tutti gli elementi necessari per esaminare in dettaglio questo programma il cui scopo è calcolare la media di *n* numeri.

L'inizio del programma è costituito da una sezione di dichiarazione delle variabili del programma.

La sezione delle istruzioni eseguibili comincia dopo la prima parola riservata INIZIO: occorre stampare un messaggio ('introdurre il numero di dati'), e poi richiedere la lettura di *n*, che rappresenta il numero di dati da leggere.

Andando avanti, interviene un'istruzione condizionale (SE ... ALLORA ... SENNÒ...), al fine di verificare se il valore di *n* è positivo. In caso affermativo, si esegue l'elaborazione desiderata: lettura dei dati, calcolo della media e scrittura del risultato. All'altra possibile alternativa (SENNÒ) corrisponde la scrittura del messaggio 'nessun dato'.

In questo programma si possono osservare diversi annidamenti di strutture, o blocchi, INIZIO ... FINE.

3.3 - Note riassuntive sulle caratteristiche del linguaggio Pascal

Cerchiamo di riassumere le caratteristiche generali del linguaggio.

3.3.1 - Dichiarazioni ed istruzioni eseguibili

Un algoritmo si compone di due tipi d'istruzioni:

— istruzione di descrizione dei *dati*, dette anche di *dichiarazione* (o definizione);
— istruzioni di descrizione delle *azioni* da compiere, dette *istruzioni eseguibili*.

Tutti i simboli utilizzati in un programma devono essere definiti in una sezione di dichiarazione. Questo potrà sembrare un vincolo, e di fatto lo è, ma nondimeno conferisce maggior rigore alla scrittura di un programma. In alcuni casi, inoltre, rende più flessibile il linguaggio, in quanto le dichiarazioni di tipo permettono al programmatore di definire dei tipi nuovi. Alcuni poi, abituati a programmare in altri linguaggi, vedranno una costrizione nella necessità di dichiarare le etichette; ma noi riteniamo che tale vincolo sia salutare, in quanto evita al programmatore di ricadere nella cattiva abitudine di usare istruzioni di salto quando e dove non sono necessarie.

3.3.2 - Le parole del linguaggio: identificatori, parole riservate, numeri, stringhe di caratteri

In un programma i dati sono rappresentati dai valori di variabili o di costanti, e vengono elaborati dalle istruzioni eseguibili.

Nel linguaggio di programmazione Pascal, variabili e costanti sono rappresentate da *identificatori*, vale a dire dai nomi con cui sono designate.

Le regole di formazione degl'identificatori sono già state viste in concreto in alcuni esempi; sono molto semplici, per cui lasciano una grande libertà di scelta.

D'altra parte, però, tutti i linguaggi di programmazione prevedono un certo numero di *parole-chiave*, o *parole riservate*, che hanno un significato ben preciso e si posso-

no usare soltanto all'interno di determinate strutture sintattiche del linguaggio. Potrete trovare l'elenco delle parole riservate del Pascal più avanti, e in appendice.

Si usano poi i *numeri*: numeri *interi* – positivi o negativi – e numeri *reali*, rappresentabili sia sotto la forma parte intera e parte decimale, sia sotto la forma mantissa con esponente.

Questa distinzione è necessaria in programmazione perchè le rappresentazioni interne dei numeri sono diverse da sistema a sistema (v. Appendice 1).

Infine, si usano anche delle entità dette *stringhe di caratteri*, che rappresentano un testo da prendere così com'è. Le stringhe sono identificate da serie di caratteri poste fra apici. Ad esempio,

'buongiorno'

è una stringa di caratteri; anche

'la scommessa di Pascal'

è una stringa di caratteri.

3.4 - Le sezioni di dichiarazione in Pascal

3.4.1 - Dichiarazione delle etichette

Le dichiarazioni delle etichette sono dei numeri che identificano determinate istruzioni di un programma. Chi ha l'abitudine di definire direttamente le etichette nel corso del programma vedrà anche qui una costrizione. Ma, se costrizione c'è, essa è stata introdotta intenzionalmente, per spingere il programmatore a non far uso di etichette, ed a costruire programmi strutturati. Ad ogni modo, al programmatore è lasciata la possibilità di definire delle etichette in caso di necessità.

3.4.2 - Dichiarazione delle costanti

Questa sezione permette di definire degl'identificatori il cui valore è costante per tutta la durata del programma. Il concetto è diverso da quello presente nella maggior parte dei linguaggi di programmazione, in cui il termine «costante» è associato in maniera diretta a numeri o a stringhe di caratteri. In Pascal una costante può essere associata ad un identificatore, e non può più essere modificata in seguito. Ad esempio,

```
const pi = 3.1416;
```

rappresenta la costante π: cioè non è un'istruzione di assegnazione (v. oltre).

3.4.3 - Dichiarazione dei tipi

Scopo di questa sezione è definire tipi di dati non standard.

Questi nuovi tipi sono definiti da un identificatore, che nel resto del programma potrà essere considerato come un nuovo tipo. Ad esempio:

```
TIPO sesso = (donna, uomo);
```

definisce il tipo *sesso* come tale da poter assumere due valori: *uomo* o *donna*.

Più avanti torneremo sull'utilità di queste dichiarazioni.

Questa sezione è facoltativa, potendosi disporre dei tipi standard, dei quali parliamo qui di seguito.

3.4.4 - Dichiarazione delle variabili in Pascal

Tutte le variabili che si usano in un'istruzione Pascal devono venir dichiarate e definite per mezzo del tipo di dato che esse rappresentano.

Il *tipo di una variabile* è definito dall'insieme dei valori che la variabile può assumere.

Ad esempio, se una variabile assume valori interi, il suo tipo sarà definito come *intero*. Siamo quindi di fronte a dichiarazioni *statiche*, che restano valide per tutto il programma.

In Pascal i dati possono essere o di tipo *scalare*, rappresentato da un insieme ordinato di valori, o di tipo *strutturato*, che definisce il metodo di strutturazione (insieme, vettore, record, flusso, etc.).

3.4.4.1. - Variabili di tipo scalare

In Pascal si possono definire delle variabili scalari particolari il cui tipo è definito da una dichiarazione di tipo. Ci sono però anche quattro tipi di variabili scalari standard: *intero*, *reale*, *carattere* e *booleano*.

Si possono poi definire i limiti di variazione di una variabile scalare indicando il più piccolo ed il più grande valore consentito: è quello che chiamiamo tipo *sottocampo* (in inglese, *subrange*).

3.4.4.2 - Variabili di tipo strutturato

In Pascal esistono quattro tipi di dati strutturati: *vettore*, *insieme*, *record* e *flusso*.

In una struttura di tipo *vettore*, tutti i componenti sono del medesimo tipo. Ciascun elemento di un vettore è selezionato mediante un *indice*, che dev'essere uno scalare e può venir calcolato. L'indice permette di accedere con il medesimo tempo ad un qualunque elemento del vettore: si tratta dunque di una struttura ad accesso casuale (random in inglese).

La struttura di tipo *insieme* definisce un insieme di valori, ricavabili da un tipo base, che dev'essere necessariamente scalare. Questa struttura corrisponde dunque all'insieme dei sottoinsiemi di valori del tipo base.

La struttura di tipo *record* è definita da componenti, detti anche campi, non necessariamente appartenenti allo stesso tipo. La selezione dei campi può essere attuata mediante un identificatore, mai mediante un indice. Gl'identificatori dei campi devono essere menzionati nella dichiarazione di record. Anche questa è una struttura ad accesso immediato.

Una struttura di tipo *flusso* è una sequenza di elementi, tutti dello stesso tipo. In un dato istante, si può accedere ad un solo elemento; si potrà accedere agli altri elementi procedendo sequenzialmente lungo il flusso. Gli elementi nuovi vengono aggiunti in coda al flusso. Si tratta dunque di una struttura ad accesso sequenziale.

3.4.4.3 - Variabili di tipo puntatore

Il linguaggio Pascal definisce un tipo particolare, detto *puntatore*. Un puntatore è una variabile che permette di far riferimento a variabili dello stesso tipo. I puntatori sono associati alle variabili strutturate, e rendono possibile la generazione di strutture di grafi orientati. Una variabile di tipo puntatore può prendere un valore particolare, il valore NULLO (o NIL), indicante che il puntatore non punta a nessun altro elemento.

3.4.5 - Dichiarazione delle procedure e delle funzioni

Un insieme d'istruzioni può essere identificato mediante un *nome*; in questo caso si parla di *procedura*. In altri linguaggi di programmazione l'insieme è detto sottoprogramma. Una procedura è, in realtà, una sequenza d'istruzioni che realizza una data elaborazione: deve perciò essere identificata mediante una *dichiarazione* di procedura, ed è utilizzabile solo nella misura in cui è caratterizzata da un certo numero di *parametri*, che rappresentano, formalmente, i dati richiesti per le elaborazioni da compiere.

Una procedura può contenere anche delle variabili locali, variabili cioè che non devono essere note alle altre procedure. È possibile definire la *portata* di una variabile: questa potrà essere locale o globale. Nel secondo caso, la sua portata abbraccia tutto l'insieme delle procedure di uno stesso programma.

In alcune realizzazioni del Pascal si possono definire dei moduli esterni, all'interno di procedure appartenenti ad una biblioteca di procedure o di sottoprocedure.

Una procedura che sia caratterizzata da un unico risultato può essere considerata una *funzione*: allora in un'espressione si potrà usare una funzione.

In Pascal intervengono i concetti di procedura e di funzione, che devono venir dichiarate e definite prima della loro utilizzazione. Torneremo ancora su quest'aspetto fondamentale del linguaggio.

3.5 - Le istruzioni eseguibili

Come si è visto negli esempi di algoritmi che abbiamo presentato, le istruzioni eseguibili sono di due tipi: istruzioni di calcolo ed istruzioni di selezione, o di controllo del programma.

In un linguaggio di programmazione evoluto, a queste istruzioni bisogna aggiungere le istruzioni d'ingresso-uscita, che in Pascal sono considerate come procedure.

In Pascal, così come in tutti gli altri linguaggi, le istruzioni di calcolo sono delle istruzioni *di assegnazione*.

Un'*istruzione di assegnazione* è formata da due parti: una parte di sinistra ed una parte di destra, separate da un operatore di assegnazione (:=). La parte a destra specifica il calcolo da eseguire, mentre la parte a sinistra indica la variabile usata per memorizzare il risultato dei calcoli specificati a destra.

Consideriamo ad esempio l'istruzione seguente:

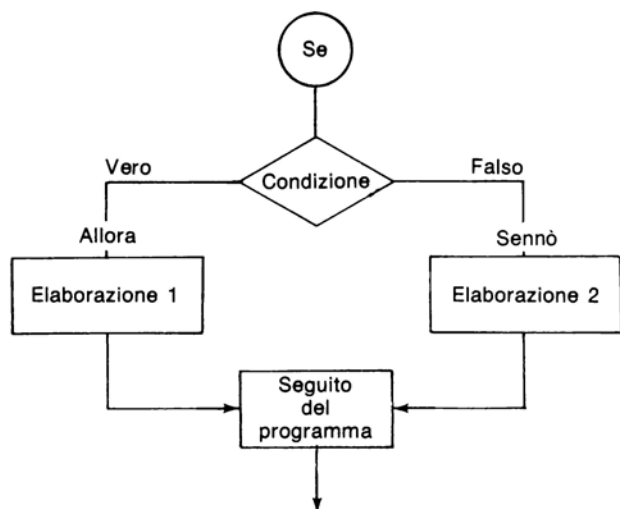
f: = a * b + c

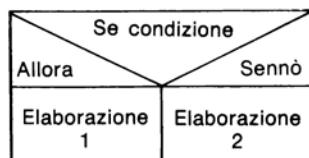
Il calcolo che viene eseguito è $a \cdot b + c$, mentre il risultato viene assegnato (memorizzato) nella variabile *f.*

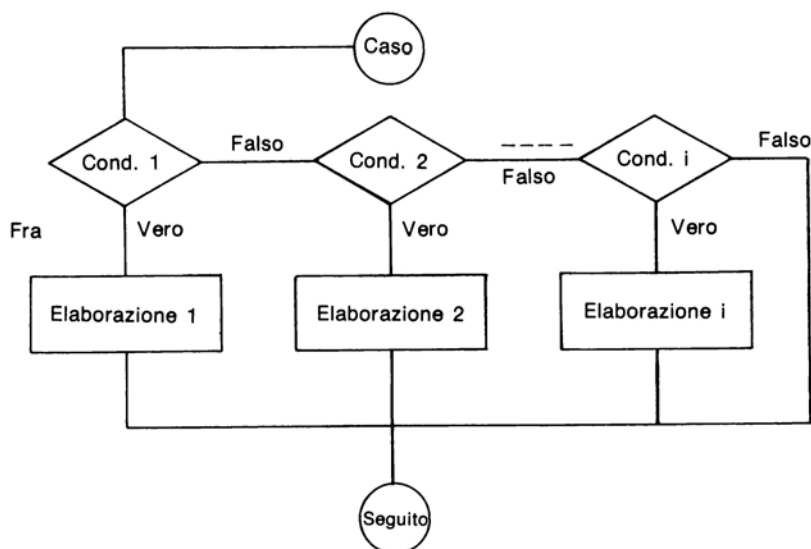
In Pascal a tipi diversi di espressioni corrispondono operatori differenti: operatori *aritmetici*, operatori *logici* (o *booleani*), operatori *relazionali* ed operatori *d'insieme*.

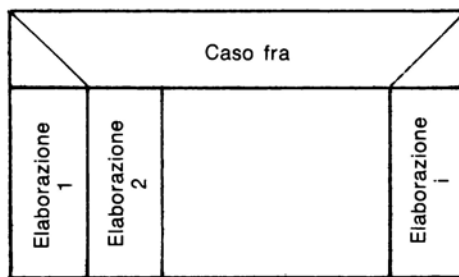
Queste istruzioni, e le regole sintattiche relative, saranno trattate in modo particolareggiato nei capitoli seguenti.

Le istruzioni *di selezione e di controllo* sono più complesse, e permettono tanto di verificare certe condizioni, alle quali corrispondono elaborazioni diverse del programma, quanto di effettuare iterazioni, o cicli di programma.

Nella categoria delle istruzioni di *test* (o *condizionali*) rientra una struttura molto comune in tutti i linguaggi di programmazione: la struttura SE ... ALLORA ... SENNÒ (IF... THEN ... ELSE in inglese), rappresentata dal diagramma di flusso che segue.

3.5.1 - Struttura di selezione semplice

Secondo la rappresentazione GNS, si ha:

3.5.2 - Struttura di selezione multipla

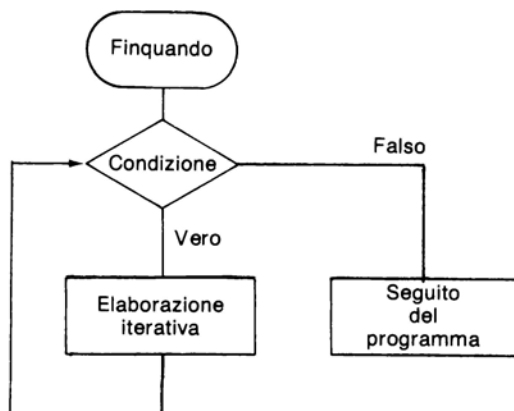
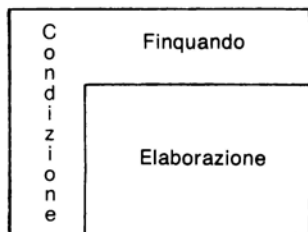
Un'altra istruzione condizionale, meno usata della precedente, è l'istruzione di selezione di un'elaborazione fra diverse possibili (più di 2). Tale struttura può indubbiamente venir programmata per mezzo di una serie di test, ma in Pascal esiste una struttura speciale, la struttura CASO... FRA... (CASE ... OF in inglese), che permette di eseguire un'elaborazione particolare per ogni valore assunto dall'espressione che segue la parola «caso». Questa struttura si può rappresentare con il solito diagramma di flusso, in questo modo:

Le condizioni *1, 2 ...i* sono espresse dai valori di un'espressione detta *selettore*.

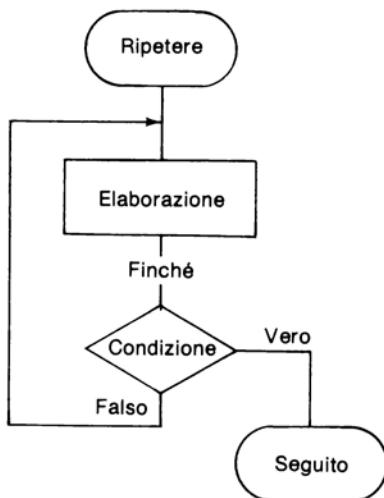
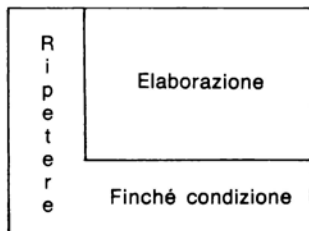
In rappresentazione GNS, la struttura è la seguente:

3.5.3 - Le strutture iterative

Per quanto riguarda le strutture iterative, in Pascal sono possibili tre strutture:

- FINQUANDO... FARE
- RIPETERE... FINCHÈ
- PER... A ... FARE

3.5.3.1 - La struttura finquando

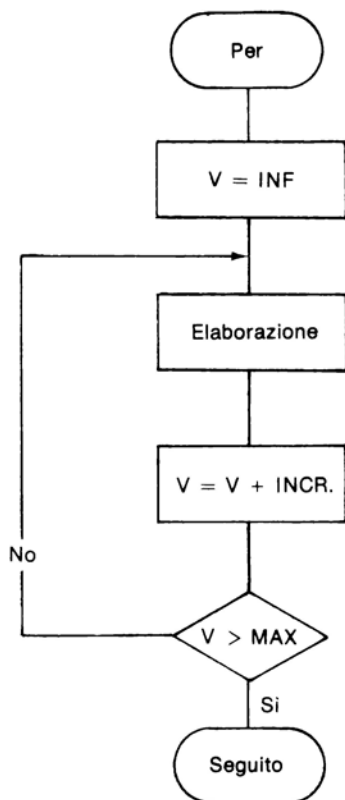
La prima struttura permette di realizzare un certo numero d'iterazioni, *fino a quando* una condizione è soddisfatta.

In rappresentazione GNS, abbiamo:

3.5.3.2 - La struttura ripetere finché

La seconda struttura permette di ripetere una sequenza d'istruzioni *finchè* una condizione non è verificata. Può essere rappresentata dal seguente diagramma di flusso:

La struttura GNS corrispondente è:

3.5.3.3 - La struttura iterativa per

L'ultima struttura consiste nel dar luogo ad un certo numero d'iterazioni mediante una variabile di controllo i cui limiti di variazione sono prefissati. È il classico ciclo di programma, utilizzato in tutti i linguaggi evoluti con DO, FOR.

Esempio

PER v: = inf A max FARE...

Questa struttura si può rappresentare con il seguente diagramma di flusso:

In Pascal l'incremento è in valore assoluto uguale all'unità.

Il ciclo PER si può effettuare anche con incrementi negativi: in questo caso la variabile di controllo decresce da un valore massimo fino ad un valore minimo.

4 - IL CONCETTO DI SINTASSI IN PASCAL

Nei paragrafi precedenti sono stati introdotti esempi d'istruzioni, che costituiscono delle frasi del linguaggio. Il testo di un'istruzione comprende parole specifiche del linguaggio (*parole-chiave*, o *parole riservate*), segni di punteggiatura, nomi specificati dal programmatore, cifre, operatori e stringhe di caratteri.

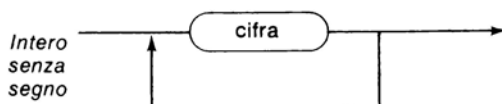
Questo complesso di simboli e parole obbedisce ad un certo numero di regole di composizione e di *sintassi*.

Pertanto imparare un linguaggio di programmazione vuol dire, prima di tutto, impa-

rarne la sintassi, anche se è chiaro che questo da solo non basta per saper risolvere un problema.

Dal punto di vista formale (cioè sintattico) il linguaggio Pascal può essere descritto in due modi: o mediante diagrammi sintattici, o mediante la forma tradizionale di Backus Naur, cioè il BNF.

Nel corso del libro ricorreremo soprattutto ai diagrammi sintattici, più espressivi della notazione BNF.

L'esempio più semplice è quello che permette di definire un *intero* senza segno. Il diagramma sintattico relativo è il seguente:

Abbiamo qui una serie indeterminata di cifre, formata minimo da una cifra.

In notazione BNF avremmo:

<intero senza segno>: = <cifra> {<cifra >}

Qui il segno: = indica una definizione, mentre i simboli fra parentesi angolari indicano la natura degli elementi che intervengono nella struttura sintattica, quegli elementi cioè che chiamiamo elementi non terminali. Gli elementi posti fra parentesi graffe indicano che la struttura corrispondente è ripetuta un numero indeterminato di volte (numero che può anche essere nullo).

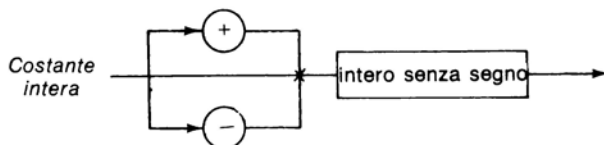
Tutti i simboli posti fra parentesi angolari devono essere definiti. Possiamo ad esempio definire <cifra> in questo modo:

<cifra>: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

La sbarra verticale indica la scelta fra vari elementi del linguaggio. Nell'esempio la scelta è fra i simboli che rappresentano le cifre decimali; queste ultime non compaiono fra parentesi angolari in quanto sono *elementi terminali* del linguaggio.

Elementi terminali sono i caratteri costituenti l'alfabeto e le parole riservate del linguaggio.

Nei diagrammi sintattici gli *elementi terminali* sono rappresentati da *blocchi ovoidali*, mentre gli *elementi non terminali* (fra parentesi angolari nella notazione BNF) da *blocchi* rettangolari.

Ad esempio, una costante intera può essere rappresentata con il seguente diagramma sintattico:

Una costante intera è definita come intero senza segno, preceduto o no dal segno + o −.

In notazione BNF avremmo:

<costante intera>: = <intero senza segno> | <segno>
<intero senza segno>

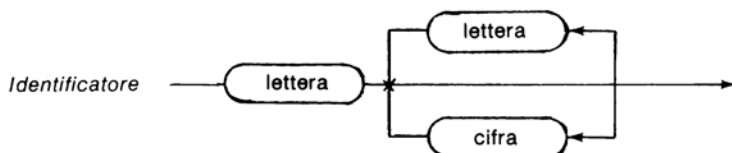
<segno>: = + | −

4.1 - Definizione di identificatore

Un identificatore è una parola scelta dal programmatore per rappresentare un oggetto del programma (nome di una variabile, costante, nome di una procedura, ecc.).

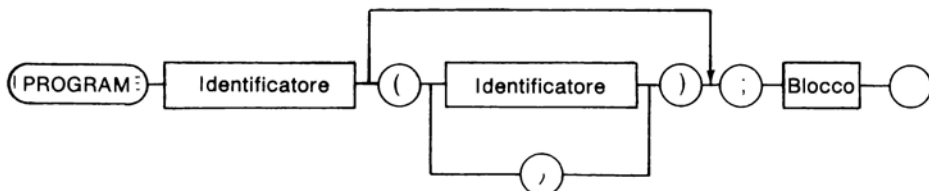
Il diagramma sintattico è il seguente:

In notazione BNF avremmo:

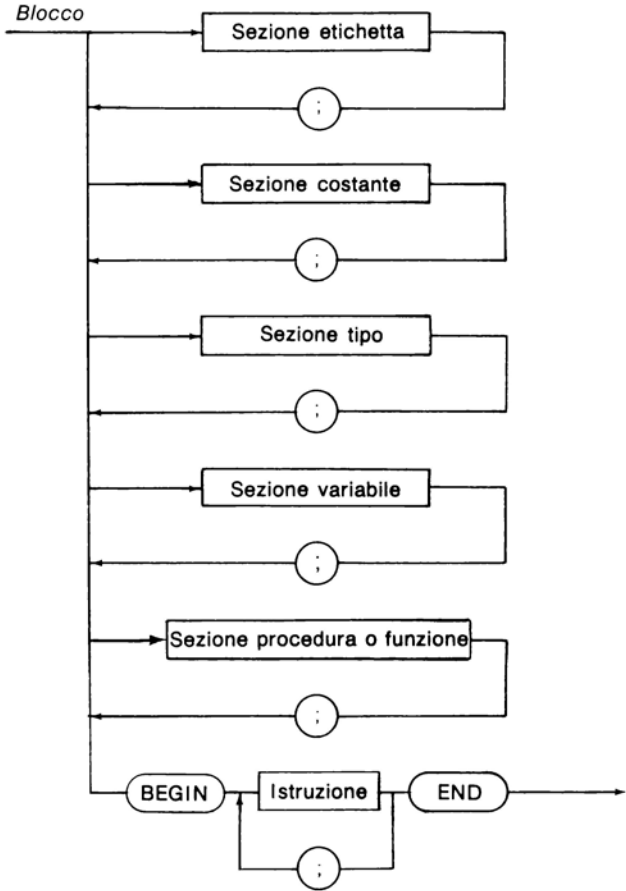
<identificatore>: = <lettera> {<lettera | cifra>}

Un identificatore deve cominciare con una lettera, e può essere seguito da un numero indeterminato di lettere o di cifre.

Nella pratica, nella maggior parte delle realizzazioni del Pascal è posto un limite al numero dei caratteri che possono differenziare due identificatori: per lo più il limite è 8.

4.2 - Definizione di programma

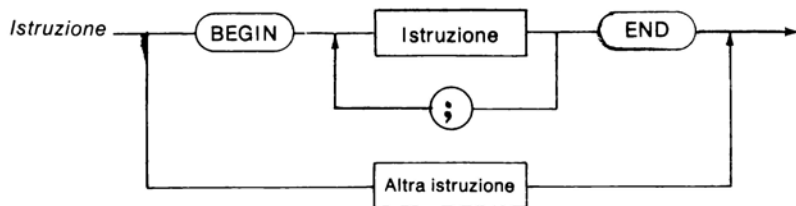
Un programma in Pascal può essere definito con il seguente diagramma:

La parte fino al carattere punto e virgola corrisponde all'intestazione del programma; la seconda parte, detta pure corpo del programma, è un blocco.

Abbiamo visto che un blocco si compone di sei sezioni; potremo pertanto rappresentarlo con il seguente diagramma:

Blocco
Sezione etichetta
;
Sezione costante
;
Sezione tipo
;
Sezione variabile
;
Sezione procedura o funzione
;
BEGIN
Istruzione
;
END

La sezione procedura (o funzione) è a sua volta scomponibile in:

– *sezione procedura*
– *sezione funzione.*

Ritroviamo qui a livello sintattico lo stesso annidamento dei blocchi di programma.

Analogamente, dall'esame della struttura dell'elemento non terminale che prende il nome d'*istruzione*, si ricava:

Lo schema fa vedere anche che un'istruzione può venir strutturata in blocchi INIZIO – FINE annidati.

Abbiamo qui una conferma della natura strutturata del linguaggio Pascal. Ne discende che è senz'altro preferibile rappresentare un algoritmo con grafi del tipo ad albero algoritmico o con grafi GNS. La programmazione in Pascal diventerà più semplice, e addirittura immediata.

Per contro i diagrammi di flusso non sono da usare, perchè darebbero luogo a programmi mal fatti.

5 - I CONCETTI FONDAMENTALI DELLA PROGRAMMAZIONE IN PASCAL

Abbiamo già presentato una serie di programmi completi in Pascal, ma nondimeno è necessario tornare sui concetti fondamentali della programmazione.

Non si può formulare un programma se non si possiede una buona conoscenza del modo in cui vanno scritti i risultati ed i messaggi: è la prima cosa da imparare, e l'illustriamo qui di seguito.

5.1 - L'istruzione di scrittura

Supponiamo di dover scrivere il messaggio 'I promessi sposi' sul dispositivo di uscita di un sistema Pascal.

L'istruzione di uscita è definita dal verbo *scrivere* (*write* in inglese) e da una serie di parametri, posti fra parentesi. I messaggi che devono essere scritti vengono posti fra apici.

Nel nostro esempio, l'istruzione sarà:

```
scrivere ('I promessi sposi')
```

o, in inglese,

```
write ('I promessi sposi')
```

Il programma più semplice che si possa scrivere è pertanto il seguente:

```
PROGRAMMA Manzoni;
INIZIO
  scrivere ('I promessi sposi')
FINE.
```

Il messaggio scritto sarà: I promessi sposi.

Supponiamo ora di dover formulare un programma che scriva il risultato di un calcolo, ad esempio il risultato della somma 4 + 5. Useremo un'istruzione quale:

```
scrivere ('Pascal scommette che 4 + 5 =', 4 + 5)
```

L'esecuzione del programma

```
PROGRAMMA calcolo;
INIZIO
  scrivere ('Pascal scommette che 4 + 5 =', 4 + 5)
FINE.
```

darà

```
Pascal scommette che 4 + 5 = 9
```

Vediamo che contemporaneamente al *risultato* della somma 4 + 5 è stato scritto un messaggio definito all'interno del programma.

I vantaggi di questa istruzione sono evidentemente molto limitati, nella misura in cui bisogna riprogrammarla tutte le volte che si esegue una semplice addizione: non si può non cambiare il messaggio al cambiare dei numeri, altrimenti si otterrebbe, ad esempio,

```
Pascal scommette che 3 + 2 = 9!
```

5.2 - I concetti di variabile e di costante

Così come s'impara a calcolare con numeri prima di passare ai simboli che rappresentano variabili possibili di assumere diversi valori, anche in programmazione si possono usare delle variabili per rappresentare valori (o dati) che cambiano ad ogni esecuzione del programma.

Possiamo ad esempio servirci di due variabili, x ed y, per rappresentare due numeri interi, o reali, e definire la variabile s con un'espressione del tipo: $s = x + y$.

In Pascal dovremo scrivere:

```
s: = x + y
```

Il simbolo : = indica che si deve eseguire il calcolo $x + y$ e memorizzare il risultato in s. Questa è un'operazione di *destinazione,* o *assegnazione.*

In Pascal tuttavia quest'istruzione, presa in se stessa, può avere significati differenti, in funzione delle dichiarazioni che la precedono. Potremo usare tanto una dichiarazione di costante quanto una dichiarazione di variabile; le due versioni del programma daranno il medesimo risultato, ma nel primo caso x ed y saranno inizializzate nella dichiarazione di costante, mentre nel secondo caso i valori di x ed y verranno assegnati durante l'esecuzione del programma.

Soltanto la prima versione permette di ottenere un programma con una sola istruzione d'assegnazione, perchè x ed y hanno in questo caso valori noti.

Il programma è allora il seguente:

```
PROGRAMMA calcolo;
COST x = 4;
      y = 5;
VAR s: intero;
INIZIO
  s: = x + y;
  scrivere ('Pascal ritiene che', x, '+', y, '=', s)
FINE.
```

Nella dichiarazione di costante si usa il segno =.

Il vantaggio di questo programma, rispetto al precedente, è che basta modificare i valori delle costanti x ed y, senza toccare il corpo delle istruzioni eseguibili, per ottenere sempre un risultato esatto. Si noti che la struttura dell'istruzione di scrittura specifica la scrittura del messaggio "Pascal ritiene che", seguito dal valore della costante x, poi dal carattere + seguito dal valore di y, poi dal carattere =, e da ultimo dal valore della somma.

All'esecuzione avremo pertanto:

```
Pascal ritiene che 4 + 5 = 9
```

Se poi ad x diamo il valore 2, e ad y il valore 3, otteniamo:

```
Pascal ritiene che 2 + 3= 5
```

Se *x* ed *y* fossero state definite come variabili, il programma sarebbe stato il seguente:

```
PROGRAMMA calcolo;
VAR x, y, s: intero;

INIZIO
   x: = 4;
   y: = 5;
   s: = x + y;
   scrivere ('Pascal ritiene sempre che', x, '+', y, '=', s)
FINE.
```

Quest'ultimo esempio prevede tre istruzioni d'assegnazione, perchè *x* ed *y* sono assimilate a delle variabili. Bisogna quindi assegnar loro un valore, per mezzo di un'istruzione eseguibile.

Inoltre, sostituendo 4 e 5 con 2 e 3 si ottiene sempre un risultato esatto. All'esecuzione avremo:

```
Pascal ritiene sempre che 2 + 3 = 5
```

Nondimeno questi due programmi presentano un inconveniente più grave: ogni modifica operata sul corpo del programma deve necessariamente ripassare attraverso un'operazione di compilazione al fine di ottenere una nuova esecuzione che porti al nuovo risultato.

Questa costrizione è assolutamente inaccettabile qualora l'utilizzatore non abbia intenzione nè di modificare nè di compilare ex novo il programma.

Nella seconda versione, ad *x* e ad *y* vengono assegnati dei valori costanti, e quindi il beneficio derivante dalla loro assimilazione a delle variabili si vanifica. Quindi, per avere una maggiore flessibilità, bisogna lasciare ad *x* e ad *y* il loro carattere di variabili, e introdurre invece un nuovo concetto: il concetto di dato.

5.3 - Il concetto di dato

Con riferimento all'esempio precedente, volendo variare *x* ed *y* da un'esecuzione all'altra, occorre che queste rappresentino dei valori forniti al sistema nel momento in cui il programma viene eseguito. Questi valori li definiamo dati.

Un *dato* è pertanto un valore che viene introdotto nel sistema dall'utilizzatore (non dal programmatore) all'*esecuzione* del programma, per mezzo di un'istruzione di *lettura*, o d'ingresso.

Il Pascal consente di richiedere l'ingresso di dati destinati ad essere letti ed elaborati dal programma.

L'istruzione *leggere* (*read* in inglese) permette d'introdurre valori che possono cambiare ad ogni nuova esecuzione.

Avremo allora il seguente programma:

```
PROGRAMMA somma;
VAR x, y, s: intero;
INIZIO
   leggere (x, y); s: = x + y;
   scrivere ('la somma di', x, '+', y, '=', s)
FINE.
```

Lo stesso programma in versione inglese è:

```
PROGRAM somma;
VAR x, y, s: integer;
BEGIN
   read (x, y);
   s: = x + y;
   write ('la somma di', x, '+', y, '=', s)
END.
```

Al momento dell'esecuzione, il programma attenderà l'ingresso dei dati *x* ed *y*.
Quando si lavora su un sistema interattivo, è molto utile far precedere l'istruzione di lettura da un messaggio che indichi che cosa si aspetta.
Supponiamo di battere 4, seguito dal ritorno carrello (tasto *return*): questo dato corrisponderà al valore di *x*. Il sistema in tal caso aspetterà un secondo valore: supponendo di battere 5, seguito dal ritorno carrello, l'esecuzione del programma proseguirà e darà il risultato previsto, cioè

la somma di 4 + 5 = 9

Si ha quindi il vantaggio che, rieseguendo il programma con i dati 124 e 385, avremo:

la somma di 124 + 385 = 509

senza dover modificare il programma.
Pertanto quest'ultima versione costituisce veramente un programma generale per eseguire la somma di due numeri qualsiasi.

5.4 - La documentazione dei programmi

Un programma semplice, come quello che abbiamo appena presentato, non ha bisogno di altre spiegazioni. Quando invece i programmi comprendono diverse decine, o anche centinaia d'istruzioni, diventa difficile per chi li legge (ed anche per il pro-

grammatore) capirne il senso. In questi casi, è sempre possibile aggiungere dei commenti, che non verranno tradotti dal compilatore, ma serviranno a documentare e/o spiegare il ruolo di ciascuna parte del programma.

I commenti s'introducono fra parentesi, con un asterisco per parte (*commento*). In alcune realizzazioni compaiono invece fra parentesi graffe: { commento }.

Il commento è un testo libero che fornisce precisazioni utili in quel punto del programma.

Esempio

```
PROGRAMMA iva;
COST iva = 0.150; (*imposta sul valore aggiunto*)
VAR presiv, preciv: reale; (*prezzo senza iva e con iva*)
     impost: reale;
INIZIO
   leggere (presiv); (*introdurre il prezzo senza iva*)
   impost: = presiv * iva; (*calcolo dell'imposta*)
   preciv: = presiv + impost; (*calcolo prezzo con iva*)
   scrivere ('prezzo senza iva =', presiv, 'iva =', impost);
   scrivere ('prezzo con iva =', preciv)
FINE.
```

Il programma non ha bisogno di altri commenti, e anche quelli esistenti sono troppi, in quanto i nomi mnemonici usati per le variabili sono già sufficientemente espliciti. Abbiamo abbondato in commenti solo per maggior chiarezza.

In generale i programmi scritti in Pascal hanno poco bisogno di commenti: se sono strutturati bene, si leggono e si comprendono facilmente. Tuttavia, quando l'algoritmo è complesso, è bene documentare i passaggi difficili. Analogamente, in alcuni casi occorre precisare il ruolo di ciascuna variabile, qualora tale ruolo non sia equivalente.

5.5 - Il Pascal come linguaggio algoritmico

I linguaggi di programmazione che permettono di trascrivere in maniera diretta algoritmi eseguibili con un calcolatore sono detti algoritmici: il Pascal è pertanto un linguaggio algoritmico che, in più, permette di programmare direttamente gli algoritmi ricorsivi.

In questo paragrafo presenteremo alcuni esempi di programmazione dei diversi tipi di algoritmi visti nel Capitolo 1.

5.5.1 - Programma di calcolo semplice

Riprendiamo il problema del calcolo della retribuzione di base, essendo noti il numero delle ore lavorative e la retribuzione oraria.

Le variabili del programma sono *h* (retribuzione oraria), *t* (tempo espresso in ore) ed *l* (retribuzione lorda). Sono variabili di tipo *reale*, e come tali verranno dichiarate, all'inizio del programma.

Il corpo del programma avrà il compito di leggere i dati *h* e *t*, calcolare il prodotto $h \cdot t$, e scrivere il risultato.

Supponiamo che il sistema sia interattivo, per cui imporremo di scrivere un messaggio prima di ogni lettura. Avremo allora:

```
PROGRAMMA retribuzione lorda;
VAR h, l, t: reale;
INIZIO
   scrivere ('retribuzione oraria =');
   leggere (h);
   scrivere ('numero di ore =');
   leggere (t);
   l: = h * t;
   scrivereln ('retribuzione lorda =', l)
FINE.
```

scriverein indica che si deve andare a capo dopo la scrittura del messaggio.

Il programma non richiede altre spiegazioni.

Supponiamo ora di voler introdurre le trattenute sociali *s*. La variabile *p* può essere considerata come una costante, perchè non varia di frequente; introdurremo poi altre due variabili per memorizzare la *s* e la retribuzione netta *n*.

Avremo allora:

```
PROGRAMMA retribuzione netta;
COST p = 0.045; (*percentuale trattenute sociali*)
VAR h, l, t, s, n: reale;
INIZIO (*ingresso dei dati*)
   scrivere ('retribuzione oraria =');
   leggere (h);
   scrivere ('numero ore =');
   leggere (t);
   INIZIO (*calcoli*)
      h: = h * t;
      s: = l * p;
      n: = l – s
   FINE; (*fine calcoli*)
scrivere ('retribuzione lorda =', l, 'trattenute sociali =', s);
scrivere ('retribuzione netta =', n)
FINE.
```

Anche questo programma è abbastanza chiaro da non richiedere ulteriori spiegazioni. Abbiamo introdotto un blocco INIZIO ... FINE supplementare per mettere bene in evidenza la parte di calcolo.

5.5.2 - Programmazione di una struttura di selezione

Se, riferendoci sempre all'esempio del paragrafo precedente, cerchiamo di risolvere il problema tenendo conto di un tetto massimo per le trattenute sociali, come si è già visto dovremo far intervenire una struttura di selezione.

Anche il parametro "tetto massimo" è una costante, che chiameremo *max* e che accluderemo alla dichiarazione di costante in questo modo:

```
COST p = 0.045; (*percentuale trattenute*)
     max = 250; (*tetto massimo trattenute*).
```

Nel corpo del programma solo il blocco di calcolo viene modificato. Avremo allora:

```
INIZIO
  l: = h * t;
  s: = l * p;
  SE s > max ALLORA s: = max;
  n: = l - s;
FINE.
```

L'alternativa SENNÒ della selezione non è usata, perché, in questo caso, non è necessario modificare *s*.

Il resto del programma rimane invariato. Lasciamo al lettore il compito di scrivere il programma completo nella versione inglese.

5.5.3 - Programmazione di una struttura iterativa

Supponiamo ora di voler realizzare l'iterazione di tutte le operazioni programmate nell'esempio precedente per applicarlo ad un insieme di persone.

Il problema principale è sapere quante iterazioni saranno necessarie, oppure determinare la condizione che provoca l'arresto delle iterazioni. Supponendo, ad esempio, che la lettura di un dato *h* nullo o negativo indichi la fine del ciclo, avremo allora il programma seguente:

```
PROGRAMMA retribuzione;
COST p = 0.045; max = 250;
VAR h, l, t, s, n: reale;
INIZIO
  RIPETERE
    scrivereln ('retribuzione oraria =');
    leggere (h);
    scrivere ('numero ore =');
    leggere (t);
    INIZIO
      l: = h * t; s: = l * p;
      SE s > max ALLORA s: = max;
      n: = l − s
    FINE;
    scrivere ('retribuzione lorda =', l);
    scrivere ('trattenute sociali =', s);
    scrivere ('retribuzione netta =', n);
  FINCHÈ h =<0
FINE.
```

Come abbiamo notato illustrando l'algoritmo, questo programma esegue un'iterazione anche per $h = 0$, cosa che è perfettamente inutile. Potremmo conservare la struttura, e testare *h* immediatamente dopo la sua lettura, ma, cosi facendo, la condizione associata a FINCHÈ diventerebbe inutile; non solo, ma questa soluzione ci costringerebbe a definire un'etichetta per uscire dall'iterazione, dando luogo alla cosiddetta struttura *ripetere uscita* (*repeat exit* in inglese). Ma su questo torneremo in seguito.

Per evitare problemi di questo tipo, occorre usare la struttura FINQUANDO, analoga a quella che abbiamo già illustrato nel paragrafo sugli algoritmi.

Avremo pertanto:

```
PROGRAMMA retribuzione;
COST p = 0.045;
       max = 250;
VAR h, l, t, s, n: reale;
INIZIO
  scrivereln ('retribuzione oraria =');
  leggere (h);
  FINQUANDO h > 0 FARE
    INIZIO
      scrivere ('numero ore =');
      leggere (t);
      l: = h * t; s: = l * p;
```

```
        SE s > max ALLORA s: = max;
        n: = l – s;
        scrivere ('retribuzione lorda =' l);
        scrivere ('trattenute sociali =' s);
        scrivereln ('retribuzione netta =' n);
        scrivereln ('retribuzione oraria =');
        leggere (h)
      FINE;
  FINE.
```

Questa versione non esegue l'iterazione per *h = 0*, ma a prezzo di una ripetizione delle istruzioni di scrittura e di lettura di *h*, all'esterno ed alla fine della struttura *finquando*.

L'ultimo modo per risolvere questo problema d'iterazione è dato dal ciclo PER, che però presuppone che il numero d'iterazioni da compiere sia noto.

Per questo motivo introdurremo un nuovo dato, che rappresenterà il numero delle persone. Avremo allora:

```
PROGRAMMA retribuzione;
COST p = 0.045; max = 250;
VAR h, l, t, s, n: reale;
     i, ind: intero;
INIZIO
  scrivere ('numero di individui =');
  leggere (ind);
  PER i: = 1 A ind FARE
  INIZIO
    scrivereln ('retribuzione oraria =');
    leggere (h);
    scrivere ('numero ore =');
    leggere (t);
    INIZIO
      l: = h * t; s: = l * p;
      SE s > max ALLORA s: = max;
      n: = l – s
    FINE;
    scrivere ('retribuzione lorda =', l);
    scrivere ('trattenute sociali =', s);
    scrivere ('retribuzione netta =', n);
  FINE;
FINE.
```

La semplicità degli esempi che precedono permette di riconoscere con chiarezza i vantaggi offerti dalle diverse strutture.

Si tenga presente comunque che la scelta dell'una o dell'altra dorà essere determinata esclusivamente in base all'algoritmo che si deve programmare; di conseguenza, prima di passare alla programmazione, è necesario verificare se l'algoritmo da programmare sia trasformabile.

5.5.4 - Trasformazione degli algoritmi

Abbiamo già illustrato il concetto di algoritmo; conosciamo d'altra parte il notissimo algoritmo di Euclide per calcolare il M.C.D. (massimo comun divisore di due numeri); questo algoritmo può essere espresso con le istruzioni seguenti:

I_1: siano due numeri interi a e b ($a > b$).
I_2: dividere a per b. Sia r il resto ($0 \leq r < b$).
I_3: *se* r è nullo, *allora* l'algoritmo termina e b è il M.C.D.
I_4: *sennò* sostituire a con b, e b con r.
I_5: proseguire da I_2.

Osserviamo subito che le cinque istruzioni non sono dello stesso tipo: I_1 è un'ipotesi, o affermazione; I_2 è un'istruzione di calcolo aritmetico; I_3 è un'istruzione condizionale (*se... allora... sennò*); I_4 è un'istruzione di trattamento di variabili; I_5 è un ordine incondizionato.

Questi differenti tipi d'istruzioni si ritrovano in tutti i linguaggi di programmazione.

Si tenga presente comunque che questa versione dell'algoritmo può venir modificata in modo da ottenere delle altre strutture.

Infatti, le istruzioni dalla I_2 alla I_4 sono equivalenti alla struttura seguente:

I_2 inizializzare r ad un valore diverso da 0.
I_3 *finquando* r <> 0 *fare*
  calcolare r = a modulo b
  sostituire a con b e b con r.
I_4 MCD= a.

Questa versione dell'algoritmo è più chiara, e può venir scritta in Pascal direttamente.

Si può anche usare una struttura siffatta:

I_2 *ripetere*
    r = a modulo b
    sostituire a con b e b con r
  finchè r = 0.
I_3 MCD = a.

Questa versione dell'algoritmo è ancora più semplice, perchè in essa non è necessario inizializzare *r* ad un valore diverso da 0; dunque è preferibile alle precedenti.

Tutte queste versioni dell'algoritmo di Euclide si basano su una proprietà aritmetica del M.C.D., espressa dalla relazione:

MCD (a, b) = MCD (b, a mod b)

che, tradotta in forma di algoritmo ricorsivo, dà luogo a:

I_2 *se* b è nullo *allora* il MCD è a
sennò MCD (a, b) = MCD (b, a mod b)

Questa forma è ancora più concisa, e permette di definire il corpo dell'algoritmo in una sola istruzione!

Il vantaggio del linguaggio Pascal è che permette di programmare in maniera diretta tutte le versioni dell'algoritmo di Euclide che abbiamo considerato.

Quindi, prima di passare alla programmazione, occorre verificare se esistono forme dell'algoritmo più concise ed eleganti, e di conseguenza più facilmente programmabili. È in questo senso che il Pascal è un linguaggio ben poco vincolante: per la sua estrema flessibilità a livello di programmazione. In Pascal, l'essenziale è pensare prima di programmare!

6 - LA REALIZZAZIONE DI UN PROGRAMMA IN PASCAL

Abbiamo visto che esistono sistemi operativi capaci di gestire le varie risorse di un calcolatore: si tratta del *monitor*, o *supervisore*.

Quando si lavora con un calcolatore che dispone di un sistema Pascal, bisogna poter dialogare con il sistema operativo, per mezzo di comandi riconosciuti dal monitor. Questi comandi non fanno parte del linguaggio Pascal in quanto tale, e di conseguenza variano da un sistema all'altro. Pertanto parleremo solo di quei comandi che costituiscono il minimo indispensabile per lo sviluppo e l'esecuzione dei programmi. Prendiamo come esempio un sistema Pascal su microcalcolatore.

I comandi, espressi mediante parole, o frasi costituite da parole-chiave, divengono attivi quando si batte il tasto di ritorno a capo, o ritorno carrello (il tasto RETURN, che in questo libro indichiamo con Ⓡ). Sui sistemi più grossi, questi comandi vengono forniti tramite schede di controllo.

Esistono poi comandi di edizione che permettono di apportate delle modifiche ad un testo già introdotto; questi comandi sono espressi da caratteri speciali.

6.1 - I comandi o istruzioni di sistema

Non ci proponiamo d'illustrare nei dettagli questi comandi, ma soltanto d'illustrarne le principali funzioni.

I comandi permettono di specificare differenti modalità di funzionamento del monitor. Nella nostra trattazione ci riferiremo ad un sistema interattivo Pascal tipo il Pascal U.C.S.D.

In generale, nel momento in cui si dà alimentazione al microcalcolatore, o in seguito all'immissione di un certo numero di codici d'identificazione nel caso di un calcolatore a divisione di tempo, il monitor segnala di essere pronto a ricevere dei comandi, inviando o un messaggio di saluto, o semplicemente un carattere speciale, come ad esempio], >, o ancora un generico carattere, che è diverso da sistema a sistema: questo carattere in gergo informatico si chiama *prompt*, cioè carattere di sollecitazione.

A questo punto l'utilizzatore può scegliere fra diversi comandi di monitor, che differiscono fra loro nei nomi mnemonici impiegati e nella sintassi. Questi comandi possono, in un certo senso, esser visti come delle istruzioni per la macchina software rappresentata dal monitor.

Un sistema Pascal prevede un numero minimo di comandi, che permettono di specificare quello che si desidera fare.

Bisogna anzitutto avere un *compilatore* (e quindi un comando che permetta di farvi riferimento) ed un *programma di link* per collegare fra loro i vari moduli di un programma. Ma su questo torneremo in seguito.

Per i sistemi Pascal è poi indispensabile anche un *sistema di gestione dei flussi.*

In un sistema interattivo, bisogna disporre di un *editor*, che permette d'introdurre e modificare i programmi.

E per finire, è indispensabile un comando di *esecuzione* dei programmi.

Con i sistemi interattivi su microcalcolatori i linguaggi di controllo hanno in generale un utilizzo molto semplice: in effetti sono trasparenti all'utente grazie alla realizzazione di sistemi ad albero di menù, che costituiscono una categoria semplificata di sistemi interattivi basati sul concetto della griglia di selezione.

Una griglia di selezione è un insieme di parole o di frasi, visualizzate sullo schermo video e selezionabili o direttamente, per mezzo di un meccanismo di selezione, o tramite tastiera. Ciascuna selezione determina la visualizzazione di un'altra griglia, che proporrà un nuovo insieme di possibili selezioni: un programma, o una procedura, associato alla selezione scelta, può venir eseguito nel frattempo. Un approccio di questo tipo risparmia all'utilizzatore la fatica di ricordare la sintassi del linguaggio di controllo.

Un sistema di questo tipo, che comporta l'uso di un'unità di visualizzazione, è disponibile sul sistema Pascal sviluppato presso l'U.C.S.D.

In esso la griglia è limitata ad una riga di comandi, che viene visualizzata nella parte alta dello schermo ad ogni fase del processo di selezione dei comandi di sistema.

La prima griglia presenta le scelte seguenti:

E(dit), *F*(ile), *R*(un), *C*(omp), *L*(ink), (e)*X*(ecute), *D*(ebug)

La selezione di un comando avviene premendo il tasto corrispondente alla lettera i-

niziale della relativa parola della griglia: si può così richiamare l'editor ed il sistema di gestione dei flussi, far partire l'esecuzione di un programma, eseguire una compilazione, collegare i moduli di un programma o mettere a punto un programma.

6.2 - L'edizione

Nei microcalcolatori sui quali lo sviluppo dei programmi è fatto in modo interattivo, è necessario disporre del programma editor, che permette d'introdurre e modificare le istruzioni, da tastiera.

Ci sono due tipi di editor, secondo che si disponga di un terminale del tipo telescrivente, che lavora riga per riga, o di un terminale video: nel primo caso, l'editor è un editor di linea, nel secondo caso è meglio utilizzare un editor di schermo. Gli editors di linea sono i più diffusi, ragion per cui molto spesso anche su schermo si lavora in modalità linea.

6.2.1 - Editor di linea

L'editor presuppone l'esistenza di un flusso di lavoro che, al momento in cui si comincia a lavorare, può essere vuoto o contenere già un testo da modificare.

Quando si è sotto il controllo dell'editor, questo segnala innanzitutto di essere pronto a ricevere un comando, inviando un carattere speciale, ad esempio un asterisco (*). Un comando si conclude battendo un tasto particolare, quale ad esempio ESC (dall'inglese *escape*: scappare), che non corrisponde a nessun carattere stampabile: nella maggior parte dei casi, l'editor invia un carattere di eco, costituito dal carattere dollaro ($). L'esecuzione di un comando, o di una serie di comandi, viene indicata battendo due volte ESC.

Gli editors di questo tipo derivano tutti da un modello, denominato TECO (correttore di testi), sviluppato nel quadro del progetto MAC presso il M.I.T. (Massachusetts Institute of Technology). Si tratta di un tipo di editor che troviamo in particolare su tutti i sistemi PDP della Digital Equipment: versioni similari esistono sulla maggior parte dei sistemi Pascal.

Quando i comandi richiedono l'introduzione del testo, si tiene conto di tutti i caratteri premuti, compreso il ritorno carrello (Ⓡ), finchè non viene premuto il tasto ESC.

La maggior parte dei comandi è in relazione con un cursore, che indica la posizione all'interno del flusso di lavoro a partire dalla quale i comandi stessi dovranno operare.

Vi è, d'altra parte, un certo numero di caratteri di controllo con un significato molto particolare: questi si ottengono premendo contemporaneamente il tasto CTRL ed un tasto alfabetico. In questo libro non illustreremo dettagliatamente tutti i comandi, o caratteri di controllo, che si usano, perché cambiano da una realizzazione all'altra; comunque, i principali comandi che si potranno incontrare sono accennati qui di seguito.

a) *Comandi d'ingresso/uscita*
Listare un certo numero di linee.
Scrivere la zona di lavoro in un flusso.
Leggere un flusso nella zona di lavoro.
Visualizzare una zona di testo a partire dalla posizione del cursore.
Lasciare l'editor.

b) *Comandi di posizionamento del cursore*
Posizionarsi all'inizio della zona di lavoro.
Far avanzare il cursore di un certo numero di caratteri o di linee.
Ricercare l'ennesima occorrenza di una sequenza di caratteri nella zona di lavoro.

Esempi

*4J$$ farà avanzare il cursore di 4 caratteri.
*8A$$ farà avanzare il cursore di 8 linee.
*BFTOTO$ = J$$ ricercherà la sequenza di caratteri TOTO, e posizionerà il cursore immediatamente prima.

c) *Comandi di edizione*
Inserire una sequenza di caratteri immediatamente dopo il cursore.
Cancellare un certo numero di caratteri dopo il cursore.
Cambiare o sostituire una sequenza di caratteri, a partire dalla posizione del cursore.
Salvare un certo numero di linee in una zona di lavoro.

Esempi

*5D$$ cancella 5 caratteri, a partire dal cursore.
*BFTOTO$ = D$$ ricerca la sequenza TOTO e la cancella.
*I$TITI$$ inserisce la sequenza TITI dopo il cursore.
*BFTOTO$ = CTITI$$ sostituisce TOTO con TITI.

Questi esempi, che valgono per l'editor del sistema Pascal-U.C.S.D., si ritrovano in forme equivalenti su altri sistemi.

Per maggiori dettagli, il lettore può rifarsi al manuale d'uso dei sistemi sui quali lavora.

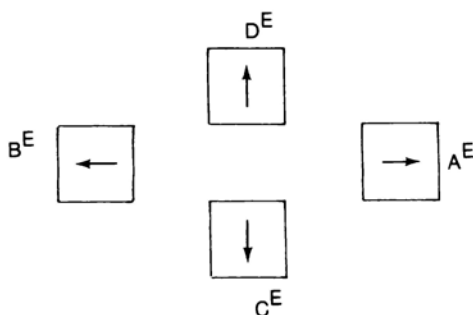
6.2.2 - Editor di schermo

Ancora su molti sistemi i terminali video sono usati dall'editor come telescriventi, cioè per lavorare riga per riga; solo con la comparsa dei microcalcolatori si sono sviluppati editors che lavorano sulla pagina visualizzata sullo schermo.

Questi editors sfruttano molto bene terminali di questo tipo, perchè permettono all'utilizzatore di constatare immediatamente l'effetto dei comandi: infatti lo schermo è come una finestra aperta sul flusso di testo. È anche visualizzato il cursore, sotto forma di un quadratino o di un carattere lampeggiante.

Nella maggior parte dei microcalcolatori le funzioni di edizione sono disponibili direttamente da tastiera, grazie alla possibilità di spostamento del cursore sullo schermo di visualizzazione.

6.2.2.1 - Movimento del cursore

I movimenti del cursore sono realizzati da tasti specifici, che ne permettono lo spostamento in tutti i sensi. Questi tasti in genere hanno impresse delle frecce, corrispondenti ciascuna ad un movimento:

Nei casi in cui questi tasti mancano, lo stesso effetto è ottenuto con un tasto speciale, generalmente ESC, che successivamente denoteremo con *E*, abbinato ad un altro tasto. Il codice ASCII di ESC è 27.

Per lo più si hanno queste combinazioni:

—A^E determina lo spostamento del cursore verso destra.
—B^E sposta il cursore verso sinistra.
—C^E sposta il cursore verso il basso.
—D^E sposta il cursore verso l'alto.

6.2.2.2 - Comandi dell'editor di schermo

Anche questi comandi possono venir precisati mediante una griglia di selezione. Ad esempio, sul sistema Pascal dell'U.C.S.D. abbiamo:

A(djust), *C*(opy), *D*(elete), *F*(ind), *I*(insert), *J*(ump), *R*(eplace), *Q*(uit), (e)*X*(change), *Z*(ap)

Questi comandi corrispondono a funzioni che si trovano su tutti gli editors di questo tipo, e che sono:

- marginatura;
- copia;
- cancellazione;
- inserimento;
- salto del cursore;
- sostituzione;
- fine della sessione di edizione;
- scambio;
- ricerca;
- cancellazione completa.

La fine di un comando è segnalata dal carattere *etx*, che può essere ottenuto dalla combinazione dei tasti CONTROL e C; l'annullamento di un comando si ottiene con il tasto ESC.

a) *Marginatura*

Questo comando permette di arretrare i capoversi rispetto al margine sinistro di un documento, di definire i margini e di giustificare le righe a destra, a sinistra o al centro rispetto ai margini.

Questo comando è particolarmente utile con un linguaggio come il Pascal, la cui struttura a blocchi può in tal modo venir rappresentata scalando i blocchi l'uno rispetto all'altro.

b) *Copia*

Questo comando permette di copiare un testo contenuto in un flusso, a partire dalla posizione del cursore, senza che venga distrutto il testo che si trova già nel flusso del lavoro.

c) *Cancellazione*

Questo comando può essere realizzato o a passi successivi, facendo avanzare il cursore di un carattere per volta e convalidando la cancellazione di ciascun carattere per mezzo di un carattere *ETX* (CONTROL + C), o globalmente, richiedendo la cancellazione di tutta una zona compresa fra un punto, definito con un comando di ricerca o d'inserimento, e la posizione attuale del cursore.

d) *Inserimento*

Questo comando permette d'inserire un testo, a partire dalla posizione attuale del

cursore. L'inserimento sarà accolto solo dopo la battitura del carattere *ETX:* è dunque possibile modificare il testo da inserire prima della convalida finale del comando.

e) *Scambio*

Per modificare un testo, carattere per carattere, a partire dalla posizione del cursore, useremo il comando di scambio. Per far questo occorre che il testo che si deve introdurre e quello da modificare abbiano lo stesso numero di caratteri. Ne consegue che il comando generalmente non va oltre il cambiamento di una o più parole per linea.

f) *Ricerca*

Dev'essere possibile richiedere la ricerca di una sequenza di caratteri all'interno di un testo: questo comando appunto permette di ricercare l'ennesima occorrenza di una serie di caratteri, qualunque sia la sua collocazione nel testo, e anche se si trova all'interno di una parola. Con questo comando si può anche specificare la ricerca di una parola singola, chiudendola fra delimitatori. A quest'operazione molto spesso segue un comando di cancellazione, d'inserimento o di sostituzione.

g) *Sostituzione*

Questo comando comporta la ricerca di lettere, o di parole, che verranno sostituite da altre lettere, o parole, tutte le volte che si troveranno. Si può anche richiedere una verifica per convalidare ciascuna sostituzione durante la ricerca nel testo.

h) *Salto*

Questo comando determina il posizionamento del cursore all'inizio o alla fine del flusso. Si può anche saltare fino a determinati marcatori introdotti nel flusso. È generalmente disponibile il comando di salto di pagina.

i) *Cambiamento di direzione*

In alcuni casi può essere utile cambiare il senso del movimento del cursore lungo il testo. Per mezzo dei caratteri < o — se ne inverte la direzione, facendolo così muovere da destra a sinistra e dal basso verso l'alto. Per tornare al senso normale (da sinistra a destra e dall'alto verso il basso) si ricorre ai caratteri > o +.

l) *Fine della sessione di edizione*

Questo comando indica che si desidera terminare la sessione di edizione. È allora necessario aggiornare il flusso di lavoro (o un altro flusso): se non si desidera nessun aggiornamento, le modifiche apportate durante l'edizione non verranno registrate.

6.3 - La compilazione

Una volta che un programma è stato introdotto mediante l'editor, e registrato in un flusso di lavoro (o in un flusso permanente), prima di poterne richiedere l'esecuzione bisogna passare attraverso la fase di compilazione.

Quest'operazione è svolta da un programma compilatore, che si può invocare con il comando *C*, nel caso che il programma sorgente sia registrato in un flusso di lavoro.

In questa fase, il compilatore traduce il linguaggio sorgente Pascal in un linguaggio oggetto intermedio, il cosiddetto codice P; contemporaneamente vengono rilevati gli eventuali errori di sintassi. Nel caso di sistemi interattivi, è possibile tornare all'editor per correggerli: essendo il Pascal un linguaggio compilato, non è possibile richiedere l'esecuzione di un programma finchè rimane anche un solo errore di sintassi!

In appendice diamo un elenco dei principali errori.

Anche per il compilatore si possono specificare dei parametri opzionali per mezzo di una griglia di selezione.

6.4 - Il collegamento dei moduli

Quest'operazione può rendersi necessaria prima della richiesta di esecuzione di un programma. In realtà il compilatore è semplicemente un traduttore, e quindi non risolve i legami esistenti fra il programma compilato e le procedure esterne, che non fanno parte della biblioteca di un sistema Pascal.

Compito del programma di link è assicurare questi collegamenti, in modo da rendere eseguibile il programma. Esso permette pure di collegare un programma a biblioteche di procedure, definite dall'utilizzatore.

Quest'operazione può venire richiesta con il comando *L* (dall'inglese *link*: legame), e può dar luogo ad errori se non è possibile risolvere determinati collegamenti sulla base delle biblioteche di procedure, standard o specificate dall'utilizzatore.

6.5 - L'esecuzione di un programma

L'esecuzione del programma può essere richiesta dopo che le due fasi precedenti si siano concluse senza errori.

Se il programma compilato risiede in un flusso di lavoro, è sufficiente invocare il comando *R* (dall'inglese *Run*: partenza), che ha il vantaggio di assicurare ugualmente il collegamento dei moduli.

Se il programma risiede in un altro flusso, in una forma eseguibile (nel qual caso ha il suffisso .OBJ), bisogna invocare il comando *X* (dall'inglese *eXecute*: eseguire).

AVVERTENZA

Quando si lavora su un sistema Pascal installato su grossi calcolatori, in modalità

di elaborazione a lotti, sono necessari altri comandi, rivolti innanzitutto all'assegnazione dei flussi e delle periferiche d'ingresso/uscita.

7 - NOTE RIASSUNTIVE

Oggetto di questo capitolo è stato l'esame degli aspetti principali della programmazione in un linguaggio evoluto. Si sono esaminati i vari tipi di algoritmo, ed i metodi strutturati di rappresentazione attualmente disponibili; a questo proposito, ripetiamo che nei capitoli seguenti non useremo più i diagrammi di flusso. I principali punti sviluppati sono stati illustrati con degli esempi.

Abbiamo poi introdotto il concetto di sintassi servendoci, come rappresentazione schematica, dei diagrammi sintattici, che continueremo ad usare nel resto del libro.

In realtà, abbiamo toccato solo le caratteristiche generali del linguaggio. Si sono poi introdotti i comandi diretti, necessari per il suo utilizzo, ed i comandi di monitor che si possono avere sui sistemi Pascal per microcalcolatori. Giova ripetere che questi comandi non fanno parte del linguaggio in quanto tale. Non esistono degli standards rigorosi, ma le funzioni che abbiamo introdotto in genere si troveranno, in una forma analoga, su tutti i sistemi. Pertanto consigliamo ai lettori alle prime armi di cercare di riconoscere sul sistema che utilizzeranno le funzioni di cui si è parlato (edizione, compilazione, etc.) e provarle con esempi semplici, prima di passare ai capitoli seguenti, nei quali si tratterà del linguaggio propriamente detto. Infatti queste funzioni sono indispensabili al fine di risparmiar tempo in fase di scrittura, modifica ed esecuzione dei programmi.

ESERCIZI

1. Prendete un programma presentato in questo capitolo. Con l'aiuto del manuale d'uso di un sistema Pascal, introducete e editate il programma. Richiedete quindi la compilazione, correggete gli eventuali errori di sintassi, assicuratevi della risoluzione dei collegamenti, e quindi eseguite il programma. Introducete i dati necessari e infine verificate che i risultati siano esatti.

   AVVERTENZA

   Si tratta di un esercizio fondamentale per i principianti. Non serve a niente precipitarsi sui capitoli seguenti, se quest'esercizio non ha dato buoni risultati.

2. Scrivere in forma algoritmica la sequenza delle operazioni che servono per eseguire la moltiplicazione e la divisione fra due numeri interi decimali. Trascrivere gli algoritmi nelle diverse forme di rappresentazione illustrate nel capitolo.

3. Problema dei quattro colori.

   L'enunciato è il seguente: se una qualunque carta geografica può essere colorata con quattro colori diversi, in modo che due stati adiacenti non siano dello stesso colore, allora scrivere 'vero', sennò scrivere 'esempio a sfavore'. Questa formulazione è un algoritmo?

   AVVERTENZA

   Parecchie generazioni di matematici non sono riuscite a dimostrarlo. Il risultato è stato provato solo di recente, grazie ad un programma!

4. Formulare un algoritmo per ordinare dei numeri in ordine crescente. Più avanti nel testo saranno date diverse soluzioni.

5. Si abbia il seguente programma in Pascal:

```
PROGRAMMA xxxx;
VAR n, y, z: intero;
INIZIO
  leggere (n);
  PER i: = 1 A n FARE
    INIZIO
      y: = n * n;
      z: = y * n;
      scrivereln (n, z, y)
    FINE;
FINE.
```

   Quali sono le variabili del programma?
   Qual è il dato?
   Quali istruzioni sono usate nel programma?
   Qual è lo scopo del programma, e quali risultati si ottengono?

CAPITOLO 3

GLI ELEMENTI FONDAMENTALI DEL LINGUAGGIO

"Gli spiriti fini, al contrario, essendo così abituati a giudicare da un solo punto di vista, sono tanto sconcertati — quando si sottopongano loro delle proposizioni delle quali non capiscono niente e per penetrare nelle quali è necessario passare per definizioni e principi così sterili, che essi non sono abituati a vedere tanto in dettaglio — che se ne allontanano e ne provano disgusto. Ma gli spiriti fallaci non sono mai né fini né geometrici; hanno dunque un ingegno retto, ma a patto di spiegar loro tutte le cose per definizioni e principi; altrimenti sono fallaci e insopportabili, perché sono retti soltanto in rapporto a principi ben chiari".

PASCAL,
Pensées

Un linguaggio, e nella fattispecie un linguaggio di programmazione, è caratterizzato da un *alfabeto*, in base al quale si possono costruire *parole* secondo regole compositive dette *regole lessicali*. Le parole di un linguaggio si riconoscono in quanto obbediscono a queste norme e sono delimitate da separatori, costituiti in generale dal carattere bianco (*blank* in inglese), espresso o con ƀ, o con ⊔, o ancora con △. Determinati caratteri possono avere un significato particolare in un linguaggio: questi caratteri sono detti *operatori*; allo stesso modo, un certo numero di parole ha in quel linguaggio un significato particolare: queste sono le *parole-chiave*, o *parole riservate*.

Dal raggruppamento delle parole, dei separatori e delle parole-chiave nascono le *frasi* del linguaggio, che nei linguaggi di programmazione prendono anche il nome di *istruzioni*.

È evidente che non tutte le frasi che si possono costruire sono corrette. La loro composizione obbedisce a certe norme: le norme della *sintassi*, che costituiscono la *grammatica* del linguaggio.

La formulazione di un linguaggio di programmazione prevede dunque la definizione arbitraria di un alfabeto e di norme lessicali; le norme sintattiche invece non sono totalmente arbitrarie, perché devono permettere di formulare degli algoritmi, e comunque devono essere scelte in modo da eliminare ogni ambiguità nella costruzione delle frasi del linguaggio.

Un linguaggio di programmazione non può permettersi di essere ambiguo. Questo, in pratica, si traduce nell'obbligo di utilizzare un certo tipo di grammatica, la cosiddetta grammatica *non contestuale*.

Le grammatiche non contestuali si caratterizzano per alcune proprietà, in base alle quali è possibile distinguere facilmente una frase corretta da una scorretta; la scrittura dei programmi traduttori (compilatori e interpreti) ne viene facilitata perché in tal modo le istruzioni possono essere tradotte senza riferimenti al contesto, e senza ambiguità.

D'altra parte, tutto ciò comporta dei vincoli a livello di programmazione: basterà ad esempio la dimenticanza di un solo carattere, o di una sola parola, per rendere scorretta un'istruzione.

In questo capitolo esamineremo le strutture del linguaggio dal punto di vista sintattico.

1 - L'ALFABETO DEL LINGUAGGIO

L'alfabeto comprende tutti i caratteri utilizzabili nelle frasi di un linguaggio: con il termine *carattere* vengono designati gli *elementi terminali di un linguaggio*.

Un linguaggio di programmazione usa un insieme di caratteri molto più vasto di quelli usati dagli alfabeti ordinari.

Nel linguaggio Pascal abbiamo i seguenti caratteri:

— le lettere maiuscole: dall'A alla Z (26 caratteri);
— le lettere minuscole: dall'a alla z (26 caratteri);
— il carattere bianco: b̸, oppure ⊔, oppure uno spazio (1 carattere);
— le cifre decimali: da 0 a 9 (10 caratteri);
— i simboli speciali, comprendenti gli operatori, i segni di punteggiatura ed i separatori:

+ — * /	operatori aritmetici
< > =	operatori relazionali
( ) [ ] { }	separatori
↑	puntatore
. , ; : '	segni di punteggiatura

Tutti questi caratteri costituiscono l'alfabeto standard del Pascal.

Partendo dall'alfabeto si possono comporre le parole del linguaggio. In programmazione per *parola* s'intende una sequenza di caratteri dell'alfabeto delimitata da caratteri separatori come il bianco, i segni di punteggiatura, gli operatori ed i delimitatori.

AVVERTENZA

La maggior parte delle realizzazioni del Pascal usa o solo le lettere maiuscole, o solo le minuscole.

In alcune realizzazioni sono disponibili dei simboli supplementari per rappresentare operatori rappresentati abitualmente da parole-chiave o da sequenze di caratteri dell'alfabeto standard. Ad esempio, per gli operatori logici,

$\neg$ rappresenta la "negazione" (not)
$\wedge$ rappresenta il "prodotto logico" (and)
$\vee$ rappresenta la "somma logica" (or)

e ancora, per gli operatori relazionali,

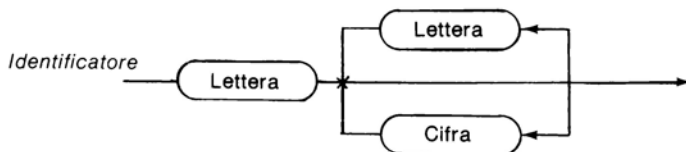
$\neq$ rappresenta l'operatore "diverso da"
$\leq$ rappresenta l'operatore "minore o uguale"
$\geq$ rappresenta l'operatore "maggiore o uguale"

2 - LE REGOLE DI FORMAZIONE DELLE PAROLE DEL LINGUAGGIO

Come si è visto nel capitolo precedente, le parole del linguaggio possono appartenere alle seguenti categorie:

- identificatori;
- numeri;
- parole-chiave, o parole riservate;
- stringhe di caratteri.

2.1 - Gl'identificatori

Le regole di formazione degl'identificatori sono molto semplici in Pascal; questo è il loro diagramma sintattico:

In altre parole, si tratta di sequenze di caratteri alfanumerici, che devono obbligatoriamente iniziare con una lettera.

Il numero dei caratteri impiegati per differenziare due identificatori può variare da una realizzazione all'altra, ma nella maggior parte dei casi è 8 (Pascal standard).

In Pascal il concetto di identificatore non è collegato al concetto di variabile, come nella maggior parte dei linguaggi di programmazione.

Le parole riservate non si possono usare come identificatori. Esistono invece degl'identificatori standard, soprattutto per i tipi standard, le funzioni e le procedure standard.

In Pascal un identificatore permette di rappresentare una variabile, una costante, un tipo, una funzione o una procedura, eventualmente definite dal programmatore.

Esempi

1) Roma Pascal Blaise bambino x1 pi due
   Yamamotokiaderate chi2 epsilon y1y2

   sono identificatori corretti.

2) integer real sin cos exp read write

   sono identificatori standard del Pascal in versione inglese.

3) intero reale leggere scrivere

   sono identificatori standard nella versione italiana.

4) San-Francisco L.S.D. 4Beatles x!2

   sono identificatori non corretti.

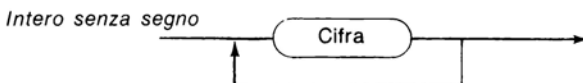
2.2. - I numeri

Il concetto di numero è chiaramente definito in Pascal, essendo prima di tutto distinto dai concetti di costante e di dato, cosa che invece non avviene nella maggior parte dei linguaggi di programmazione.

In informatica si distinguono attualmente due tipi di numeri: i numeri interi e i numeri reali. Questa distinzione è dovuta semplicemente a vincoli di ordine tecnologico, che fanno sì che i due tipi di numeri non siano rappresentati nello stesso modo all'interno del calcolatore.

Per i numeri interi si ha una rappresentazione in base 2 (binaria), con una convenzione per i numeri negativi (il complemento a due).

Per i numeri frazionari si usa la rappresentazione convenzionate detta a virgola mobile.

Un numero *intero* senza segno è rappresentato dal diagramma seguente:

Esempi

1) 124 è un numero intero.

2) 1984 è un numero intero.

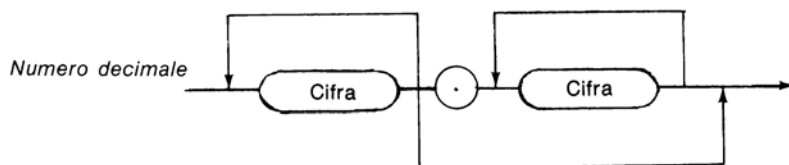
Vedremo che, in pratica, il campo di variazione degl'interi è limitato, dal momento che le parole-macchina hanno dimensioni fissate da ragioni costruttive (8, 16, 32 bits).

I numeri *decimali* sono rappresentati con una parte intera ed una parte decimale, separate da un *punto*, non da una virgola.

Esempi

1) 1284.78 è un numero decimale.

2) 3.14 è un numero decimale, detto anche reale dato che in realtà non è che un'approssimazione del numero reale π.

Il diagramma sintattico dei numeri decimali è il seguente:

Esiste poi una forma più generale di rappresentazione dei numeri reali legata alla rappresentazione a virgola mobile.

2.2.1 - La rappresentazione a virgola mobile

All'interno del calcolatore i numeri o i valori reali vengono codificati mediante la cosiddetta rappresentazione a virgola mobile.

Il principio è semplice, essendo basato su una rappresentazione esponenziale. Ad esempio, il numero 454.87 può venir rappresentato in diverse forme, quali

4.5487×10^2
45487×10^{-2}
0.45487×10^3

Moltiplicando il numero per una potenza di 10, positiva o negativa, si può spostare il punto all'interno del gruppo delle cifre significative: di qui il nome di virgola (o punto) mobile.

In questa rappresentazione le cifre significative costituiscono la *mantissa*: per sapere qual è il numero basta conoscere la potenza di 10 ad essa associata.

Esiste poi una forma, detta *normalizzata*, nella quale la mantissa è compresa fra 0.1 e 0.99...: relativamente all'esempio precedente, la forma normalizzata è 0.45487×10^3. Questa forma è usata dai sistemi Pascal in fase di uscita dei risultati. Allora, per rappresentare un numero a virgola mobile, viene inserito il carattere E come separatore fra la mantissa e l'esponente.

Esempi

1) 0.314E+1 = 3.14

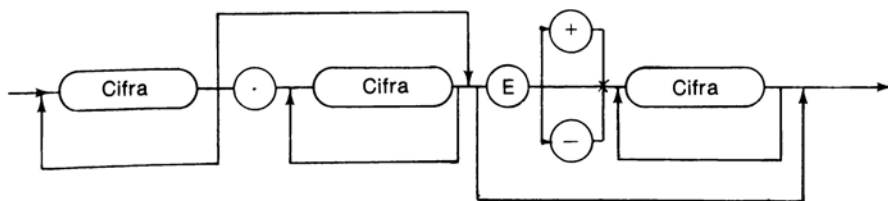
2) 0.14768E+3 = 147.68

3) O.5E+0 = 0.5

4) 0.25E–2 = 0.0025

Questa notazione, di facile interpretazione per l'utenza scientifica, non incontra invece il favore dell'utenza gestionale, in quanto non si addice all'edizione di documenti come fatture, estratti di conti bancari, etc.

Concludendo, in Pascal il diagramma sintattico di un numero è il seguente:

Numero senza segno

Questo diagramma ha validità generale, perché permette di rappresentare sia i numeri interi e decimali, sia i numeri reali scritti in notazione mobile.

Esempi

1) 44 6744 17.6 12E4 85E6 123E–7
sono numeri scritti correttamente in Pascal.

Invece

2) .5 3,14 18:4 1/2 VI E–5 4.567.923
non sono scritti correttamente.

2.3 - Le stringhe di caratteri

Abbiamo già incontrato delle stringhe di caratteri quando abbiamo visto la stampa di un testo.

La norma che regola le stringhe di caratteri vuole che queste siano racchiuse fra apici (').

Esempi

'Alberto' 'Roma' 'Pascal' '125,32'
'esempio di stringa' 'a + b' 'buongiorno'

Nel Pascal standard la lunghezza delle stringhe non è sottoposta ad alcun limite. Il testo posto fra apici può essere quel che si vuole, purchè costituito da caratteri che facciano parte dell'alfabeto.

Un caso particolare è l'apostrofo. Tutti gli apostrofi contenuti nel testo costituente la stringa devono essere duplicati.

Esempi

1) 'quest''oggi'
   rappresenta la stringa *quest'oggi.*

2) 'l''apostrofo e''duplicato in papa'''

AVVERTENZA

Le stringhe di caratteri, che hanno una funzione assolutamente primaria all'interno del corpo di un programma, non vanno confuse con i commenti, la cui sola funzione è quella di documentare il programma. Ricordiamo che un commento, in Pascal, viene introdotto fra parentesi graffe oppure fra i caratteri * e *, come si può vedere nel programma seguente:

```
PROGRAMMA stringa;
INIZIO
   (*questo programma scrive una stringa*)
   scrivere ('una stringa in Pascal');
FINE.
```

2.4 - Le parole-chiave o parole riservate del linguaggio

Le parole-chiave del linguaggio Pascal sono scritte in inglese: qui le elenchiamo in italiano, con la relativa versione inglese a destra.

Parole chiave di dichiarazione

COST	CONST
VAR	VAR
TIPO	TYPE
VETTORE	ARRAY
RECORD	RECORD
INSIEME	SET
FLUSSO	FILE
ETICHETTA	LABEL
IMPACCATO	PACKED
DI	OF

Parole - chiave condizionali, o di selezione

SE ... ALLORA ... SENNÒ	IF ... THEN ... ELSE
CASO FRA	CASE OF

Parole - chiave iterative

RIPETERE ... FINCHÈ	REPEAT... UNTIL
FINQUANDO ... FARE	WHILE ... DO
PER ... A	FOR ... TO
DISCENDENTE	DOWNTO

Parole - chiave generiche

PROCEDURA	PROCEDURE
FUNZIONE	FUNCTION
PROGRAMMA	PROGRAM
INIZIO	BEGIN
FINE	END

Parole - chiave di controllo

CON	WITH
ANDARE A	GOTO

Parole - chiave operatori

E	AND
O	OR
DIV	DIV
MOD	MOD
IN	IN
NON	NOT

Parola - chiave di nullo

NULLO NIL

Le parole riservate sono considerate come *delimitatori*.

Gli altri delimitatori sono i simboli speciali dell'alfabeto: operatori, segni di punteggiatura, etc.

I delimitatori vanno distinti dai *separatori*, che sono il carattere bianco (*blank* in inglese), il ritorno a capo (rappresentato con Ⓡ in questo libro), ed i commenti. Il numero di separatori che si possono avere fra due parole, o due delimitatori, è indifferente.

2.5 - Alcuni esempi

1. Sono corretti in Pascal i seguenti identificatori?

   n1, Ciccio, pa, r$, luna, 1k, b100, b44, 0SS117, m+1, Nabucodonosor, New York, Breizh, Luigi XV

2. Sono corretti i seguenti numeri?

   15 24,15 72.54 3^2 –14 ±31.1 .55 0.68E+2 1.24,E–4 2/3/79

3. Scrivere delle stringhe costanti di caratteri. Si possono usare parole-chiave all'interno delle stringhe di caratteri?

Soluzioni

1. sì sì sì no sì no sì sì sì no sì no sì sì.
2. sì no sì no sì no no sì no no.
3. 'basta inventarle'
   Sì, si possono usare parole - chiave all'interno delle stringhe di caratteri.

3 - LE REGOLE DI PROGRAMMAZIONE NEL LINGUAGGIO PASCAL

Finora abbiamo presentato degli esempi, e definito gli elementi fondamentali del linguaggio: l'alfabeto, gl'identificatori, i numeri, le parole riservate, etc. Impariamo ora a servirci di questi elementi, ed a creare frasi, o istruzioni, corrette dal punto di vista sintattico, delle frasi cioè che rispettino le norme grammaticali del linguaggio. Queste regole di programmazione potranno sembrare al neofita precise, o addirittura rigide; è quindi indispensabile conoscerle bene, ed usarle nel modo corretto.

La seconda fase consiste nella formulazione di un programma che abbia un senso! E qui si opera ad un livello che non dipende più dal sistema, nè dal linguaggio, ma dal

cervello dell'utilizzatore, oltre che da quello che viene definito come la semantica, o il senso, dell'oggetto della programmazione.

Presenteremo degli esempi semplici, scelti in base al loro valore pedagogico ed alla loro capacità di dare un'idea delle possibilità del linguaggio. Da questi esempi il lettore dovrebbe acquisire una certa familiarità con un programma in Pascal, fermo restando in ogni caso che soltanto la *pratica* di un linguaggio consente di acquisirne una sufficiente padronanza.

Il lettore non dovrà accontentarsi di leggere dei programmi, nè di ricopiarli su un calcolatore; al contrario, dovrà assolutamente affrontare problemi nuovi e sforzarsi di risolverli con le sue sole forze. Ci auguriamo che quanto verremo esponendo possa permettere al lettore di apprendere tutti i concetti utili alla programmazione, e gli fornisca tutti quegli attrezzi conoscitivi che servono per usare il linguaggio studiato.

Nel corso di questo capitolo torneremo in maniera dettagliata sui diversi tipi d'istruzioni: le dichiarazioni, le istruzioni aritmetiche, le istruzioni di test, le istruzioni iterative, le istruzioni d'ingresso/uscita.

3.1 - I tipi d'istruzioni in Pascal

A parte i testi fra parentesi graffe o fra asterischi, che sono commenti o osservazioni, distinguiamo fondamentalmente cinque categorie d'istruzioni:
- le istruzioni di dichiarazione;
- le istruzioni di assegnazione, che comprendono a loro volta le istruzioni aritmetiche e le istruzioni di manipolazione d'insiemi;
- le istruzioni di selezione: IF ... THEN ... ELSE (SE ... ALLORA ... SENNÒ); la selezione multipla: CASE ... OF (CASO ... FRA);
- le istruzioni iterative: FOR ... (PER ...), WHILE (FINQUANDO), REPEAT (RIPETERE);
- le procedure d'ingresso/uscita: read, write (leggere, scrivere).

3.2 - Struttura di un'istruzione

Il linguaggio Pascal non ha bisogno del formato scheda perforata.

Nel Pascal standard, si assume di solito come supporto delle istruzioni la riga di 80 caratteri; comunque un'istruzione non termina a fine riga.

Un'istruzione può essere costituita da più frasi, ciascuna terminante con un punto e virgola (;), ma tutte le istruzioni terminano con un punto e virgola.

Si possono quindi avere più istruzioni, o frasi, su una medesima riga.

4 — LE DICHIARAZIONI IN PASCAL

Come si è già visto, in Pascal esistono cinque sezioni di dichiarazione. Le prime quattro (dichiarazione delle etichette, dichiarazione delle costanti, dichiarazione dei

tipi e dichiarazione delle variabili) saranno esaminate in questo capitolo. La sezione di dichiarazione delle funzioni e delle procedure invece sarà studiata più avanti.

4.1 - Dichiarazione delle etichette

In Pascal un'etichetta è costituita da un numero intero seguito da un'istruzione eseguibile. Il numero identifica in modo univoco l'istruzione, e dev'essere seguito da un delimitatore, il carattere due punti (:). Consideriamo, ad esempio, la seguente istruzione:

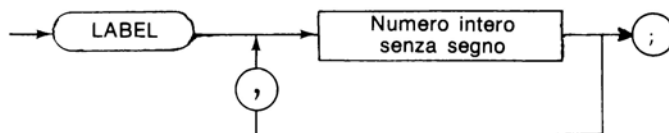
```
1: scrivere ('qui siamo all''etichetta 1');
```

Quando il programma darà il controllo a quest'istruzione, per mezzo di un'istruzione di diramazione (v. oltre), verrà scritto il messaggio corrispondente.

Si tenga presente che, a differenza della maggior parte dei linguaggi di programmazione meno recenti, nei quali le etichette sono viste come «naturali», o addirittura indispensabili, il linguaggio Pascal esige che *ciascuna* etichetta venga definita in una dichiarazione preliminare; e questo volutamente, per impedire al programmatore di abusarne.

Regola fondamentale è dunque chiedersi, prima di definire un'etichetta, se è veramente indispensabile. Torneremo su questo punto quando studieremo le istruzioni di selezione e le istruzioni di diramazione (il famoso GOTO, cioè ANDARE A).

Il diagramma sintattico di questa dichiarazione è il seguente:

Etichetta

Una dichiarazione di questo tipo è riconosciuta dalla parola riservata LABEL, cioè ETICHETTA nella versione italiana, seguita da una serie di numeri interi senza segno, separati da virgole.

Esempi

```
LABEL 1, 12, 56;
ETICHETTA 5, 80, 6;
```

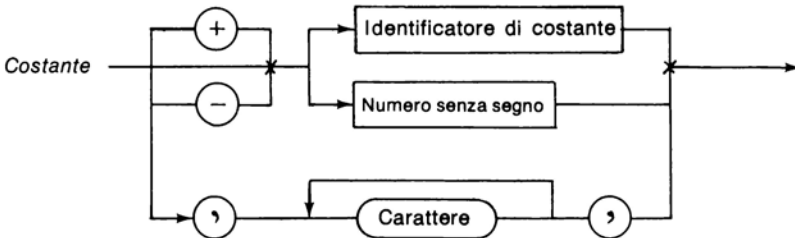
In pratica, nel Pascal standard un'etichetta può contenere fino a quattro cifre. Soltanto le etichette dichiarate saranno considerate valide, e pertanto non c'è nessuna

ragione per non impiegare numeri consecutivi: in alcuni casi i numeri usati potranno riferirsi ad una semantica personale del programmatore.

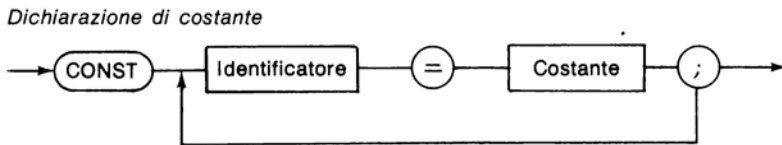
4.2 - Dichiarazione delle costanti

In Pascal esistono due tipi di costanti: le costanti numeriche e le costanti costituite da stringhe di caratteri.

Una costante numerica può essere definita o direttamente, mediante un numero dotato di segno, o per mezzo di un identificatore, definito nella sezione di dichiarazione delle costanti. Una costante formata da una stringa di caratteri è costituita da un testo posto fra due apici.

Il diagramma sintattico di una costante è pertanto il seguente:

(C'è poi un'altra costante, detta NIL (NULLO), che ha un significato molto particolare: la ritroveremo associata al concetto di puntatore).

La dichiarazione di costante permette di associare un identificatore ad una costante numerica o ad una costante formata da una stringa di caratteri.

Il diagramma sintattico della dichiarazione di costante è il seguente:

Si tratta dunque di una dichiarazione nella quale la parola riservata, CONST, è seguita da una serie d'identificatori il cui valore è specificato dalla costante che si trova dopo il segno di uguale (=).

Esempi

```
1)      CONST pi = 3.14; uno = 1; dieci = 10;
              iva = 0.150;
              epsilon = 1E–10;
              sottolineatura = '__________';
              titolo = 'risultati';
```

Questa dichiarazione obbedisce ad alcune regole:

- La parola riservata CONST può comparire una sola volta all'interno di un programma o di una procedura.
- La dichiarazione CONST è facoltativa. Se c'è, deve venire dopo la dichiarazione di etichetta, e precedere la dichiarazione di tipo e di variabile.
- Il carattere = indica proprio l'uguaglianza. È quindi diverso dal simbolo di assegnazione (:=).
- L'identificatore di una costante non può cambiare di valore nel corso del programma cioè non deve mai trovarsi alla sinistra del segno: = in un'istruzione di assegnazione.
- Non si può usare lo stesso identificatore per definire due costanti, un tipo o una variabile.

2) Esempio di programma in cui compaiono delle costanti:

```
PROGRAM cerchio;
CONST pi = 3.14159;
       due = 2; titolo = 'cerchio di raggio=';
VAR raggio, circonferenza, superficie: real;
BEGIN
  (*calcolo della circonferenza e della superficie di un cerchio*)
  read (raggio); writeln (titolo, raggio);
  circonferenza: = due * pi * raggio;
  superficie: = pi * raggio * raggio;
  writeln ('Circonferenza =', circonferenza, 'superficie =' superficie);
END.
```

4.3 - Dichiarazione dei tipi

Il concetto di tipo è legato a quello di dato: infatti il tipo rappresenta l'insieme dei valori che un dato può assumere. I dati vengono elaborati per mezzo di variabili: pertanto sarà l'identificatore della variabile associata ad esser definito come avente un determinato tipo.

Tutti i linguaggi di programmazione utilizzano il concetto di tipo, ma in generale il programmatore non ha la possibilità di definire nuovi tipi.

In Pascal i tipi standard non hanno bisogno di venir definiti con una dichiarazione; ma per converso il programmatore ha la possibilità di definirne di nuovi, specifici del proprio problema.

4.3.1 - I tipi standard

I tipi standard sono quattro:

- il tipo intero;
- il tipo reale;

— il tipo booleano;
— il tipo carattere.

I tipi *intero* e *reale* sono già stati illustrati in esempi precedenti. In Pascal vengono espressi rispettivamente con i due identificatori standard *integer (intero)* e *real (reale)*.

Si tenga presente che i valori assunti dalle variabili associate a questi tipi non corrispondono agl'insiemi infiniti che incontriamo in matematica, avendo infatti dei limiti superiori ed inferiori fissati in fase di progettazione del sistema, e legati alla lunghezza delle parole-macchina che si usano per rappresentare quei numeri.

Se, ad esempio, gl'interi vengono rappresentati con *n* bits, l'intervallo di rappresentazione degl'interi, positivi e negativi, sarà:

$$-2^{n-1} \leqslant \text{interi} \leqslant +2^{n-1} - 1$$

(v. Appendice 1).

Allora per $n = 16$, come nella maggior parte dei microcalcolatori, avremo:

$$-32768 \leqslant \text{interi} \leqslant +32767$$

L'insieme degl'interi disponibili può dunque essere relativamente limitato. Bisognerà quindi star bene attenti, quando si usa il tipo intero, a non superare i limiti previsti dal sistema col quale si lavora.

Anche per i reali esistono dei limiti, ma molto più estesi, e dipendenti dalla rappresentazione con la quale sono espressi i numeri a virgola mobile internamente al sistema (v. Appendice 1).

Il tipo reale si usa quando bisogna eseguire calcoli precisi, ma si tenga presente che qui il termine "reale" in realtà coincide con il termine "razionale". I numeri irrazionali, come $\sqrt{2}$, ed i numeri trascendenti, come π ed e, vengono sempre espressi con delle approssimazioni. Del resto il tipo reale può essere utilizzato quando si lavora su valori interi che non rientrino nel campo disponibile per gl'interi.

Il tipo intero è necessario in alcune istruzioni (parametri di controllo di un ciclo PER), per l'indice di un vettore ed in alcune funzioni standard, quali *pred* e *succ* (v. oltre). Infine esiste un identificatore di costante, detto *maxint* nella versione standard, che rappresenta il massimo valore assoluto che un intero può assumere nel sistema utilizzato.

4.3.1.1 - Il tipo booleano

Il tipo *booleano* viene a volte definito, in altri linguaggi, tipo logico, e fa riferimento al concetto di valore logico, o valore booleano (dal nome del matematico Boole, che su di esso ha sviluppato un'algebra), cioè ad un valore che esprime la verità logica: *vero* o *falso*.

Ci sono due costanti booleane, rappresentate da identificatori standard: *true* (*vero*) e *false* (*falso*).

Una variabile di tipo booleano può assumere l'uno o l'altro di questi valori. Più avanti illustreremo gli operatori e le funzioni booleane.

Quello che segue è un semplice esempio di definizione di un ciclo infinito in cui si è utilizzata una costante booleana.

```
PROGRAM persempre;
BEGIN
  WHILE true DO
    write ('Pascal ha scritto per l''eternità'');
END.
```

Avremmo potuto introdurre anche una variabile booleana:

```
PROGRAM persempre;
VAR semprevero: boolean;
BEGIN
  semprevero: = true;
  WHILE semprevero DO
```

È chiaro che la lettura sarebbe più piacevole in Pascal in versione italiana:

```
INIZIO
  semprevero: = vero;
  FINQUANDO semprevero FARE
    scrivere ('Pascal ha scritto per l''eternità'');
FINE.
```

4.3.1.2 - Il tipo carattere

Anche questo è un tipo standard in Pascal. Abbiamo già visto le costanti formate da stringhe di caratteri, dette anche stringhe letterali, che sono costituite da una sequenza di caratteri fra apici; il tipo *carattere* invece è formato da un unico carattere.

L'insieme dei caratteri disponibili può cambiare da un sistema all'altro, ma in ogni caso comprende l'insieme dei caratteri alfanumerici, più il carattere bianco (*blank* in inglese).

L'identificatore standard per rappresentare questo tipo è *char* (*car* nella versione italiana).

Il valore di una variabile di tipo carattere è un carattere che si possa scrivere e stampare: una tale variabile non può quindi contenere che un solo carattere. Vedremo tuttavia che è possibile definire delle variabili costituite da una serie di caratteri, per mezzo di vettori.

Con le variabili - carattere si possono usare gli operatori di confronto.

Esempio

```
PROGRAMMA ricercaparola;
COST bianco = ' ';
VAR carattere: car;
INIZIO
  RIPETERE
    leggere (carattere)
  FINCHÈ carattere = bianco;
FINE.
```

Questo programma permette di leggere una sequenza di caratteri *finché* viene trovato il carattere spazio indicante la fine di una parola.

4.3.1.3 - Il tipo stringa di caratteri

Nel Pascal standard questo tipo non è disponibile, perché il tipo carattere permette di definire soltanto variabili costituite da un solo carattere.

La versione U.C.S.D. propone un'estensione che permette di definire le stringhe di caratteri come un tipo standard. Nel Pascal standard, invece, bisogna definire dei vettori di sequenze di caratteri (per la dichiarazione del tipo vettore, vedi oltre).

Nella versione U.C.S.D. l'identificatore associato alle stringhe di caratteri è *string* (*stringa*).

Esempi

1)

```
PROGRAM stringa;
CONST verbo = 'essere'; coniug = 'non';
VAR frase: string;
BEGIN
  frase: = 'to be or not to be';
  writeln (frase);
  frase: = coniug;
  writeln (verbo, 'o', frase, verbo);
END.
```

Il programma definisce delle costanti costituite da stringhe di caratteri, denominate *verbo* e *coniug*, ed una variabile di tipo stringa di caratteri (*frase*): usando una variabile, si ha la possibilità di assegnare ad essa diversi valori di stringhe di caratteri nel corso del programma.

Il programma permette innanzitutto di assegnare alla variabile *frase* la frase 'to be or not to be', e di stamparla; poi a questa variabile viene assegnato il valore di una costante (*coniug*), costituita da una stringa di caratteri. L'ultima istruzione fa vedere

che è possibile scrivere delle stringhe di caratteri espresse o come costanti, o come stringhe letterali, o ancora come variabili: la frase scritta è 'essere o non essere'.

2) Quando in una variabile è memorizzata una stringa di caratteri, si può aver bisogno di accedere a dei caratteri singoli della stringa: questo è realizzabile grazie alla possibilità di indicizzare una variabile-stringa mediante una costante o una variabile intera posta fra parentesi quadre.

```
PROGRAM caratt;
VAR frase: string;
    car: char;
    n: integer;
BEGIN
  frase: = 'sono  le 4' (*2 spazi fra sono e le *)
  car: = frase [1] (*car contiene il 1° carattere: s*)
  n: = 4; car: = frase [n];
  frase [5]: = car; n: = 3; car: = frase [n];
  frase [4]: = car; frase [1]: = 'e';
  frase [2]: = 'r'; frase [3]: = 'a';
  writeln (frase);
END.
```

Il programma verifica la possibilità di assegnare i caratteri di una stringa ad una variabile, e viceversa.

Come risultato, al presente *sono* è sostituito l'imperfetto *erano*. La scrittura della variabile *frase* al termine del programma darà quindi: 'erano le 4'.

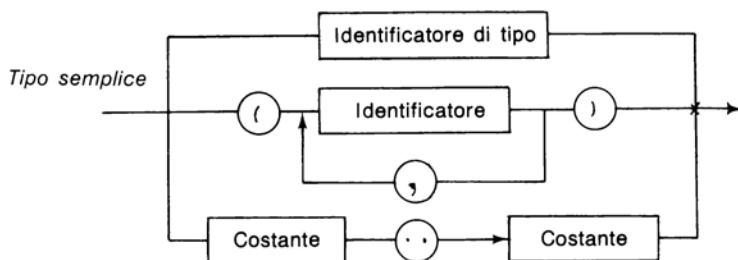
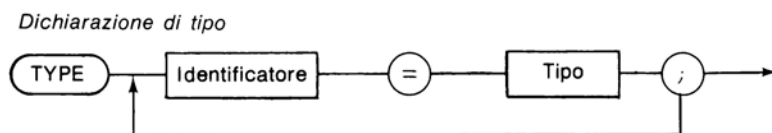
4.3.2 - I tipi non standard

La dichiarazione di tipo permette di specificare dei tipi particolari, propri del problema in esame.

In questo capitolo ci occuperemo solo dei tipi elementari: scalari e sottocampi di scalari. Questi tipi vengono definiti dal programmatore, e valgono unicamente all'interno del programma, o della procedura, nei quali sono stati definiti: non sono assolutamente necessari, ma permettono di rendere più chiaro il programma, e, soprattutto, di garantire che le variabili corrispondenti non contengano altri valori oltre a quelli associati al tipo definito.

Un tipo *scalare* è definito da un insieme ordinato di valori che possono venir specificati mediante identificatori.

Un *sottocampo* di scalare è definito da un limite inferiore ed un limite superiore, definiti all'interno dell'insieme associato.

Il diagramma sintattico dei tipi elementari è il seguente:

L'ultima riga corrisponde ad un sottoinsieme di scalare, detto sottocampo. Allora il diagramma sintattico di una dichiarazione di tipo è:

Esempio

TYPE	misura =	(grandissimo, grande, abbastanzagrande, medio, piccolo, abbastanzapiccolo, piccolissimo);
	sesso =	(femminile, maschile);
	logico =	(vero, falso);
	mese =	(gennaio, febbraio, marzo, aprile, maggio, giugno, luglio, agosto, settembre, ottobre, novembre, dicembre);

Vedremo fra non molto che i tipi scalari possono venir associati a delle variabili, alle quali sono applicabili funzioni ed operatori relazionali. Si vedrà che non si può usare lo stesso identificatore per due diversi tipi. Ciascun insieme che definisce un tipo dev'essere disgiunto dagli altri, cioè l'intersezione è nulla.

L'esempio che segue è dunque inesatto:

```
TYPE primavera = (marzo, aprile, maggio, giugno);
        estate = (giugno, luglio, agosto, settembre);
```

perché l'identificatore *giugno* compare in entrambi i tipi definiti.

4.3.2.1 - Il tipo sottocampo

È invece possibile definire dei tipi che siano sottoinsiemi ordinati di insiemi già definiti (standard o no). Sono i cosiddetti tipi *sottocampo.* Gli unici sottocampi che non si possono definire sono quelli di tipo reale.

Esempio

```
TYPE lunghezzamese = 0 ... 31;
     inverno = (gennaio ... marzo);
```

È ovvio che dev'esserci stata la definizione del tipo *mese*, non riportata qui. *Inverno* è definito come tipo sottocampo di *mese:* in questo caso l'inclusione dev'essere rigorosa.

Si possono inoltre definire sottoinsiemi di tipo carattere:

```
TYPE lettera = 'a' ... 'z';
     cifra = '0' .... '9';
```

Nel secondo caso, le cifre sono state considerate come caratteri; se invece avessimo scritto:

```
TYPE decimale = 0 ... 9;
```

si tratterebbe di numeri interi decimali di una sola cifra.

Le norme che regolano i tipi sottocampo sono semplici:

- Per definire un tipo *sottocampo* occorre che il tipo ad esso associato sia stato definito nella stessa procedura, oppure che sia un tipo standard non reale.
- Il primo valore specificato (limite inferiore) dev'essere disposto prima dell'ultimo valore (limite superiore) all'interno dell'insieme associato al tipo sottocampo.

La definizione seguente sarebbe dunque inesatta:

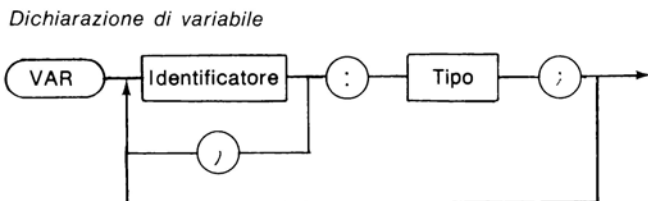
```
TYPE estate = (settembre ... luglio);
     eta = 120 ... 0;
```

L'impiego dei tipi sottocampo permette di definire con estrema precisione il campo di validità delle variabili che si utilizzano: alcune istruzioni di assegnazione, pur corrette dal punto di vista sintattico, si possono rivelare ineseguibili a causa dei tipi associati dalle variabili dell'istruzione.

In seguito torneremo su questi problemi.

4.4 - Dichiarazione delle variabili

Questa sezione di dichiarazione è necessaria per scrivere un algoritmo nel quale intervengano delle variabili: il suo ruolo è infatti definire ciascuna variabile con il suo tipo.

Il diagramma sintattico è il seguente:

Esempi

1) VAR a, b, c: real;
i, j: integer;
caratt: char;
bandiera: boolean;

2) Si possono anche definire variabili che corrispondano a tipi non standard, o direttamente, per mezzo dei tipi sottocampo, o riferendosi ad un tipo già definito nella dichiarazione di tipo, come è mostrato qui di seguito:

```
TYPE giorni = (lunedì, martedì, mercoledì, giovedì, sabato, domenica);
      weekend = sabato ... domenica;
VAR decimale: 0 ... 9; binario: 0, 1;
    giorno: giorni;
    vacanza: weekend;
    caratt: 'a' ... 'z';
```

In questo esempio le variabili *giorno* e *vacanza* sono definite con tipi dichiarati; invece le variabili *decimale*, *binario* e *caratt* sono definite direttamente, come sottoinsiemi di tipi standard, interi o carattere. Tutto questo è perciò equivalente all'insieme delle due dichiarazioni

```
TYPE cifra = 0 ... 9; bit = 0, 1;
      alfabeto = 'a' ... 'z';
VAR decimale: cifra; binario: bit;
    caratt: alfabeto;
```

3) Anche i tipi sottocampo di un tipo non standard possono venir definiti direttamente. Supponendo che il tipo *giorni* sia stato definito, possiamo definire la variabile

```
VAR lavorativo: lunedì ... venerdì;
```

Le norme per la dichiarazione delle variabili sono le seguenti:

- La parola riservata VAR può comparire non più di una volta in un programma o in una procedura.
- Si possono invece scrivere tutte le linee di dichiarazione necessarie, ed anche più frasi di dichiarazione di uno stesso tipo.
- La dichiarazione di variabile non permette di assegnare un valore iniziale ad una variabile.
- Il nome ed il tipo di una variabile sono permanenti.

Ogni assegnazione che non corrisponda al tipo definito provoca un errore di esecuzione; questo vale anche per il mancato rispetto del limite inferiore o superiore di un tipo sottocampo.

Consigliamo di usare nomi di variabile che richiamino il significato o la funzione delle variabili stesse: in caso contrario è opportuno introdurre un commento che ne indichi il significato.

4.4.1 - Il concetto di ambiente di una variabile

Si è visto che uno stesso identificatore non può essere utilizzato, in una dichiarazione di variabile, per rappresentare due variabili diverse. D'altro canto sappiamo che il Pascal è un linguaggio strutturato a blocchi. L'*ambiente* di una variabile, ossia la possibilità di riferirsi a quella variabile, in un'istruzione, come ad un oggetto identificato in modo univoco, con un tipo anch'esso definito in modo univoco, concerne solo le istruzioni del blocco nel quale la variabile in questione è stata definita.

È possibile ridefinire la variabile con un tipo diverso in un altro blocco: per questo diciamo che l'ambiente di una variabile è locale rispetto al blocco nel quale è stata definita.

Se due blocchi B1 e B2 sono tali per cui B2 è annidato in B1, le variabili di B2 saranno locali per B2, ed inaccessibili in B1. Invece una variabile dichiarata in B1 è accessibile in B2, a patto che non sia stata dichiarata ex novo in B2.

L'ambiente di una variabile concerne tutto il blocco nel quale la variabile è stata definita, a meno che non sia ridefinita in un blocco più interno. Consideriamo, ad esempio, il seguente programma schematico:

```
blocco B1
..........
VAR x, y: real;
..........
   blocco B2
   ..........
   VAR x: integer;
```

In questo caso, *y* è una variabile reale in tutto il blocco B1, mentre la variabile *x* è reale all'interno di B1, ma non all'interno di B2, in cui è intera.

4.5 - Dichiarazione delle funzioni e delle procedure

Questa sezione è facoltativa, e viene obbligatoriamente prima della sezione delle istruzioni eseguibili.

Per il momento non la esamineremo, perché la sua struttura presuppone l'esistenza di un blocco, e quindi di una sezione d'istruzioni eseguibili contenute nel blocco. Allora bisogna prima studiare quest'ultima sezione.

5 - LE ISTRUZIONI ESEGUIBILI IN PASCAL

Abbiamo già visto alcune strutture di istruzioni eseguibili.

In questo capitolo riesamineremo tutte le strutture delle istruzioni in modo completo e sistematico. Le istruzioni eseguibili si trovano nell'ultima sezione del programma, l'unica obbligatoria: questa sezione inizia con la parola riservata BEGIN (INIZIO) e termina con la parola riservata END (FINE).

La parola riservata END (FINE) indica soltanto la fine di una struttura e, in particolare, la fine della sezione delle istruzioni eseguibili. La fine del programma è invece indicata dal carattere *punto* (.).

Quindi un programma termina con la parola riservata END (FINE) seguita dal *punto*. Quando il compilatore incontra questa sequenza, la compilazione di arresta, e si torna al monitor.

AVVERTENZA

Con alcuni sistemi e in particolare con il sistema Pascal dell'U.C.S.D., si può richiedere il ritorno all'editor quando viene segnalato un errore di compilazione: ciò naturalmente presuppone che si lavori con un sistema interattivo.

5.1 - Le istruzioni di assegnazione

In Pascal l'istruzione di assegnazione è applicabile a tutti i tipi di variabile, tranne le variabili strutturate di tipo flusso (v. oltre).

Le istruzioni di assegnazione possono venir definite per le variabili di tipo intero, reale, booleano, carattere, e anche per le variabili appartenenti a tipi specifici, definiti nella sezione di dichiarazione di tipo.

La forma generale di quest'istruzione è:

identificatore di variabile: = espressione;

Il simbolo: = è l'*operatore di assegnazione*: esso indica che l'espressione che compare al secondo membro viene valutata, e che il suo valore viene attribuito (o assegnato) alla variabile il cui identificatore è specificato al primo membro.

A tal fine occorre che il tipo della variabile che compare al primo membro sia *compatibile* con il tipo dell'espressione che compare al secondo membro. Spiegheremo in seguito il concetto di compatibilità fra tipi e vedremo che essa è più estesa della condizione d'identità fra tipi.

Se l'espressione e la variabile sono dello stesso tipo, hanno necessariamente tipi compatibili fra loro.

Partiremo dall'esame delle istruzioni di assegnazione di tipo aritmetico, che sono quelle di gran lunga più usate, ed assolutamente indispensabili in tutti i linguaggi di programmazione evoluti.

5.2 - Le istruzioni di calcolo aritmetico

Nei capitoli precedenti abbiamo visto dei semplici esempi d'istruzioni di calcolo aritmetico. In programmazione le istruzioni aritmetiche permettono di specificare quei calcoli che chiamiamo solitamente algebrici.

Esempio

```
a: = b + c;
c: = 2 * pi * r;
```

Un'istruzione di questo tipo si caratterizza per il fatto che al primo membro deve comparire un *identificatore* di variabile numerica, al secondo membro un'*espressione aritmetica*. I due membri sono separati dal simbolo: =, che non è un segno di uguaglianza: è detto infatti simbolo di *attribuzione*, o di *assegnazione*.

Questo segno indica infatti che il calcolatore deve eseguire il calcolo dell'espressione che si trova al secondo membro dell'istruzione, e che il risultato ottenuto dev'essere assegnato, e memorizzato, nella variabile che definisce il primo membro dell'istruzione.

Pertanto, in programmazione, un'espressione come

```
x: = x + 2;
```

è corretta. Non va considerata come un'equazione: infatti esprime la somma del contenuto della variabile *x* e della costante letterale numerica 2, e l'assegnazione del risultato al primo membro, che in questo caso è la variabile *x*. In altre parole, l'espressione permette di sommare 2 ad *x*, per cui è chiaro che il contenuto originario di *x* va perduto. Se ad esempio il contenuto di *x* prima dell'esecuzione dell'istruzione è 5, il risultato finale sarà 5 + 2, cioè 7. Il nuovo valore di *x*, in seguito all'esecuzione, sarà dunque 7.

L'istruzione di assegnazione permette di modificare il contenuto di una variabile in modo dinamico, proprietà che tornerà particolarmente utile con gli algoritmi di tipo iterativo basati su formule ricorsive.

Esempio

Supponiamo di voler eseguire la somma dei primi *n* numeri interi.

L'algoritmo può essere espresso con una formula ricorsiva; infatti, supponendo nota la somma dei primi $i - 1$ numeri, cioè s_{i-1}, otteniamo la somma dei primi *i* numeri sommando *i* ad s_{i-1}.

Allora

$$s_i = s_{i-1} + i$$

L'algoritmo sarà il seguente:

I_1 leggere n
I_2 s = 0; i = 1
I_3 *finquando* i $\leq$ n *fare*
I_4 s = s + i
I_5 i = i + 1
I_6 *fine* iterazione
I_7 scrivere s

Il corrispondente programma in Pascal è il seguente:

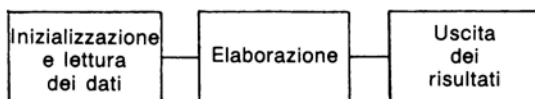
```
PROGRAMMA som;
VAR s, n, i: intero;
INIZIO
   leggere (n); s: = 0; i: = 1;
   FINQUANDO i < = n FARE
   INIZIO
      s: = s + i; i: = i + 1;
   FINE;
   scrivereln ('somma dei primi', n, 'numeri =', s);
FINE.
```

Le istruzioni di assegnazione del programma sono *s: = 0* ed *i: = 1*, che assegnano dei valori iniziali alle variabili *s* ed *i*; *s: = s + i*, che accumula le somme parziali dei primi *i* numeri; infine *i: = i + 1*, che permette di passare al successivo numero intero.

Pertanto il programma è formato da tre parti:

- una parte statica, corrispondente all'inizializzazione delle variabili ed alla lettura dei dati;
- una parte dinamica, corrispondente alla memorizzazione dei risultati intermedi ottenuti via via per *s* ed *i*: quest'ultima parte corrisponde alla formula ricorsiva associata all'algoritmo;
- una parte finale, anch'essa statica, che stampa i risultati.

Questa struttura si ritrova tipicamente in tutti gli algoritmi, e di conseguenza in tutti i programmi. In conclusione si ha:

- *Prima parte:* inizializzazione delle variabili e/o lettura dei dati.
- *Seconda parte:* dinamica, coincide con l'elaborazione da effettuare.
- *Terza parte:* è la parte finale, e coincide con l'uscita dei risultati e la fine dell'algoritmo.

5.2.1 - I vari tipi di espressioni aritmetiche

Un'istruzione aritmetica ha la forma

v: = espressione aritmetica

dove *v* è una variabile e: = è il simbolo di assegnazione.

Un'*espressione aritmetica semplice* è caratterizzata da una sequenza d'identificatori di variabili o di costanti, separati da operatori aritmetici.

Gli operatori aritmetici in Pascal sono sei:

+	addizione
–	sottrazione
*	moltiplicazione
/	divisione
DIV	divisione intera
MOD	operatore modulo (resto di una divisione intera)

Per tradurre una generica espressione matematica, è sufficiente renderla lineare mediante gli operatori che abbiamo elencato. Si tenga presente che va tenuto conto di tutte le operazioni, anche di quelle non esplicitate nell'espressione iniziale.

Esempi

1) L'espressione $ab + cd$ si traduce con

a * b + c * d

2) Analogamente $b^2 - 4ac$ si traduce con

b * b – 4 * a * c

5.2.1.1 - Regole di precedenza, o di gerarchia, fra gli operatori

Consideriamo l'espressione

a * b/c * d

In assenza di regole di precedenza, può essere interpretata in due modi:

$$\frac{a \cdot b}{c \cdot d} \text{ oppure } \frac{a \cdot b}{c} \cdot d$$

In realtà l'espressione in questione dovrà essere valutata come segue:

– tutti gli operatori hanno priorità 1;
– le operazioni vengono eseguite da sinistra verso destra.

Si avranno allora le seguenti fasi:

$$a \cdot b$$

$$\frac{a \cdot b}{c}$$

$$\frac{a \cdot b}{c} \cdot d$$

per cui l'espressione matematica corrispondente è

$$\frac{a \cdot b \cdot d}{c} \quad \text{e non} \quad \frac{a \cdot b}{c \cdot d}$$

Consideriamo ora un'espressione nella quale compaiano tutti gli operatori:

a/b + c – d DIV e * f MOD 2

L'espressione sarà valutata eseguendo prima l'operazione

$$\frac{a}{b}$$

poi l'operazione

d ÷ e * f modulo 2

ed infine

$\frac{a}{b}$ + c − d ÷ e * modulo 2

In Pascal nelle espressioni non si può ricorrere a quegli accorgimenti tipografici che sono abituali in matematica. È necessario quindi stabilire alcune regole che diano luogo ad un'interpretazione unica di un'espressione data, definire cioè una gerarchia degli operatori aritmetici, detta anche *ordine di precedenza degli operatori.*

Definiamo per prime le regole riguardanti le espressioni aritmetiche *semplici*, nelle quali non intervengono parentesi nè funzioni.

Regola 1

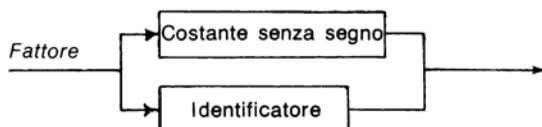
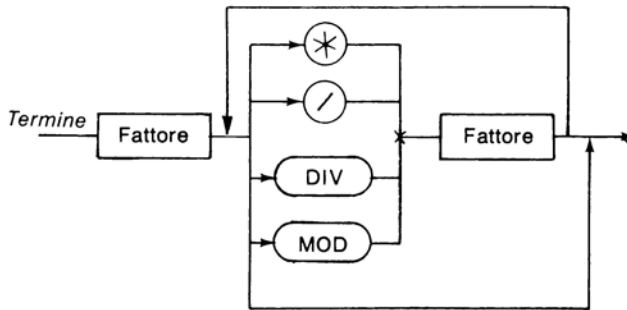
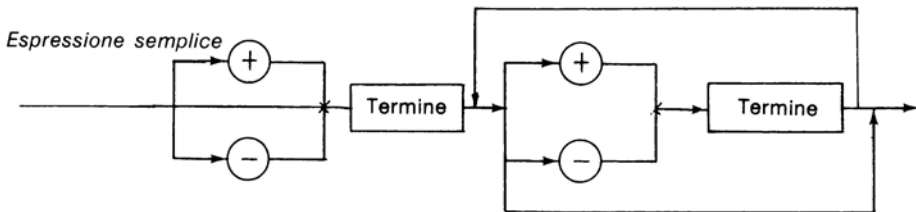
Un'espressione aritmetica semplice si valuta procedendo da sinistra a destra, ed eseguendo prima le operazioni di moltiplicazione (*), divisione (/ o DIV) e modulo (MOD); poi, sempre da sinistra a destra, le operazioni di addizione (+) e sottrazione (−).

Si ha dunque la seguente gerarchia degli operatori aritmetici:

—	moltiplicazione	*	priorità 1
	divisione	/DIV	"
	modulo	MOD	"
—	addizione	+	priorità 2
	sottrazione	−	"

5.2.1.2 - Rappresentazione sintattica delle espressioni aritmetiche

Siamo volutamente partiti con l'esporre le regole di precedenza secondo i modi tradizionali; ma avremmo potuto introdurle altrettanto bene da un punto di vista puramente sintattico.

In questo modo definiremo innanzitutto il *fattore* mediante il diagramma seguente:

poi il *termine*:

e infine l'*espressione semplice*:

Questi diagrammi riassumono compiutamente le regole di precedenza enunciate prima, dal momento che il termine è definito sulla base di fattori separati dagli operatori a priorità più alta, mentre l'espressione semplice è definita sulla base di termini separati dagli operatori a priorità più bassa.

5.2.1.3 - Introduzione delle parentesi

In alcune espressioni matematiche più complesse è necessario introdurre delle parentesi. Si tenga presente che in programmazione non è consentito rappresentare le espressioni complesse con accorgimenti tipografici di "accatastamento" come questo:

$$\frac{a + \frac{b}{c}}{c + \frac{d + e}{k}}$$

Di qui la necessità d'introdurre delle parentesi. Con riferimento all'esempio precedente, il numeratore è equivalente a

a + b/c

e il denominatore a

$$c + d/k + e/k$$

L'espressione $c + d + e/k$ non rappresenta in modo corretto il termine

$$c + \frac{d + e}{k}$$

Usando invece le parentesi, possiamo scrivere:

$$c + (d + e)/k$$

Analogamente useremo le parentesi per indicare la divisione dell'espressione al numeratore per l'espressione al denominatore. L'espressione finale corretta sarà pertanto

$$(a + b/c)/(c + (d + e)/k)$$

Se ne deduce la regola seguente:

Regola 2

Le espressioni fra parentesi vengono valutate con una priorità superiore rispetto a tutte le altre operazioni.

Possiamo dire quindi che le parentesi hanno priorità 0.

OSSERVAZIONI

Alcune espressioni possono essere programmate in più di un modo. Ad esempio, l'espressione

$$\frac{a \cdot b}{c \cdot d}$$

è programmabile in tre modi:

— prima forma:
    (a * b)/(c * d)

— seconda forma:
    a * b/(c * d)

che è un'espressione corretta, perchè esegue ($c \cdot d$) e poi $a \cdot b$, che viene infine diviso per $c \cdot d$.

— terza forma:

a * b/c/d

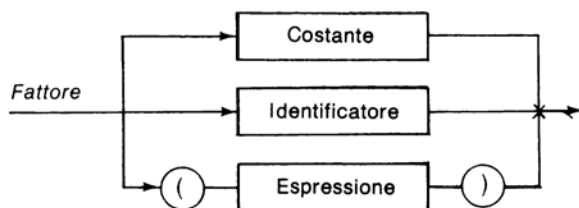
nella quale non si hanno parentesi, e purtuttavia il risultato è esatto. Infatti si esegue $a \cdot b$, che viene poi diviso per c, cioè

$$\frac{a \cdot b}{c}$$

che viene diviso per d, cioè

$$\frac{a \cdot b}{c \cdot d}$$

Vediamo dunque che la programmazione di una formula di tipo frazionario non richiede in ogni caso le parentesi. Tuttavia è pur sempre meglio introdurre delle parentesi inutili piuttosto che scrivere una forma errata. Quindi i principianti sono autorizzati ad introdurre un numero eccessivo di parentesi; a patto però che siano certi della validità dell'espressione.

Dal punto di vista sintattico, un'espressione fra parentesi viene rappresentata come un fattore:

Nel diagramma è operante la regola 2, per la quale le espressioni fra parentesi vengono valutate per prime.

5.2.2 - Le funzioni matematiche standard del Pascal

Alcune espressioni matematiche utilizzano funzioni matematiche standard: radice quadrata, esponenziale, logaritmo, funzioni trigonometriche, etc.

Una funzione è allora caratterizzata dal *nome*, che è un identificatore standard del linguaggio, seguito da un'espressione aritmetica fra parentesi. Questo evidentemente presuppone che l'espressione fra parentesi rientri nel campo di definizione della funzione. Ad esempio l'espressione associata ad una radice quadrata dovrà essere

sempre positiva o nulla. Il programmatore dovrà stare bene attento a che questa condizione sia soddisfatta, perchè in caso contrario un programma, corretto dal punto di vista sintattico, potrà dar luogo ad errori di esecuzione. Di seguito diamo un elenco delle funzioni matematiche standard che si usano solitamente in Pascal.

abs (x)	Valore assoluto di x
arctan (x)	Arcotangente di x
cos (x)	Coseno di x
exp (x)	Esponenziale di x
ln (x)	Logaritmo neperiano di x
sin (x)	Seno di x
sqr (x)	Elevamento al quadrato di x
sqrt (x)	Radice quadrata di x

Altre funzioni che danno valori numerici sono:

trunc (x)	Questa funzione dà la parte intera di un numero reale
round (x)	Questa funzione arrotonda un numero reale al numero intero più prossimo.
ord (x)	Questa funzione dà il numero d'ordine di x all'interno dell'insieme dei valori dello stesso tipo.
chr (x)	Questa funzione, il cui parametro è intero, fornisce il carattere che ha numero d'ordine x.
succ (x)	Questa funzione può essere definita per un qualunque tipo scalare, eccettuati i reali. Dà il successivo valore di x all'interno dell'insieme del tipo associato.
pred (x)	È la funzione simmetrica della precedente. Permette di ottenere il valore che precede nell'insieme del tipo associato.

Nel Capitolo 5 torneremo sull'uso delle funzioni, e parleremo di alcune funzioni che non compaiono nell'elenco riportato nella tabella, e che sono abitualmente usate in altri linguaggi di programmazione, e cioè:

- tan (x) — Tangente di x
- cot (x) — Cotangente di x
- Log (x) — Logaritmo decimale

In ogni caso le eventuali funzioni supplementari sono pur sempre calcolabili partendo dalle funzioni standard. Ad esempio,

tan (x) = sin (x)/cos(x)
cot (x) = 1/tan (x)
Log (x) = ln (x)/ln (10)

Non è necessario disporre di queste funzioni per poterle utilizzare. Più avanti vedremo qualche esempio di funzioni derivate.

5.2.2.1 - Introduzione delle funzioni standard in un'espressione aritmetica

Gli argomenti delle funzioni matematiche possono essere delle espressioni aritmetiche poste fra parentesi. Ritroviamo, a questo livello, la priorità 0 per la valutazione delle espressioni corrispondenti: in più, nello stesso momento in cui l'espressione è valutata, viene calcolata la funzione.

Da questo deriva la regola seguente:

Regola 3

In un'espressione aritmetica le funzioni matematiche sono valutate con priorità 0, allo stesso modo delle espressioni fra parentesi.

Esempi

1) Si abbia l'espressione

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

programmata nella forma

```
(-b + sqrt (b * b - 4 * a * c))/(2 * a)
```

Si calcolano prima l'espressione fra parentesi e la funzione, e cioè

```
sqrt (b * b - 4 * a * c)
```

quindi

```
-b + sqrt (b * b - 4 * a * c)
```

e poi ancora

```
2 * a
```

Infine si divide la prima espressione per la seconda.

2) Si è visto che l'operazione di elevamento a potenza non esiste in Pascal: bisogna allora ricorrere alla funzione esponenziale.
Si abbia l'espressione

$$r = c \cdot \frac{i}{1 - (1 + i)^{-n}}$$

che rappresenta il calcolo della cifra da restituire a fronte di un capitale c, rimborsabile in n anni ad un tasso d'interesse dell'i%.

L'espressione può venire comunque programmata, sapendo che $a^x = e^{x ln(a)}$, per cui

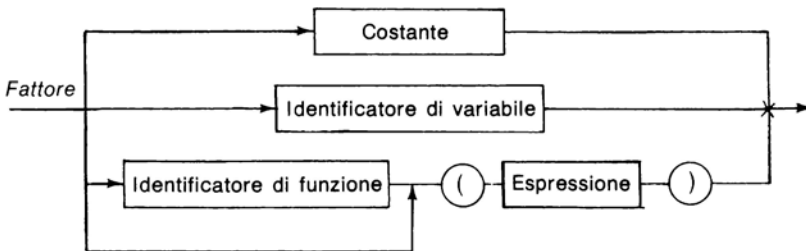
$$(1 + i)^{-n} = e^{-n ln(1+i)}$$

L'espressione corrispondente in Pascal è la seguente:

r = c * i / (1 − exp (−n * ln (1 + i)))

Da questo esempio si vede che è possibile l'annidamento di più funzioni. La valutazione di quest'espressione sarà fatta in questo modo: $ln(1 + i)$, poi $nln(1 + i)$, poi $e^{-nln(1+i)}$, poi ancora $1 - e^{-nln(1+i)}$. A questo punto si esegue il prodotto $c \cdot i$, ed infine si divide quest'ultimo per l'espressione al denominatore già calcolata.

5.2.2.2 - Rappresentazione sintattica

In un'espressione aritmetica le funzioni sono considerate come *fattori*. Si ha pertanto:

Nel diagramma sintattico ritroviamo l'ordine di precedenza enunciato nella regola 3.

5.2.3 - Note riassuntive sulle istruzioni aritmetiche

Abbiamo visto come un'istruzione aritmetica è un'istruzione di assegnazione del tipo

nome della variabile: = espressione aritmetica

Pertanto le regole sintattiche sulle espressioni aritmetiche si possono così riassumere:

Regola 1

La precedenza tra gli operatori nella valutazione delle espressioni aritmetiche è:

I) Parentesi e funzioni matematiche. Si calcolano per prime le parentesi e le funzioni più interne.

II) Moltiplicazione, divisione (* / DIV), e l'operazione modulo, nell'ordine in cui si presentano procedendo da sinistra a destra.

III) Addizione e sottrazione (+ −), nell'ordine in cui si presentano leggendo l'espressione da sinistra a destra.

Il fatto di usare in un programma espressioni di questo tipo non elimina la necessità di obbedire ad un certo numero di regole, le cosiddette regole semantiche, di cui le due principali riguardano la conoscenza preliminare dei valori delle variabili e l'omogeneità del tipo delle espressioni.

Regola 2

— Tutte le variabili utilizzate in un'istruzione aritmetica devono avere un valore al momento in cui l'istruzione viene eseguita.
— Le espressioni che intervengono come parametri delle funzioni devono corrispondere a valori possibili per la funzione.

Regola 3

— Tutte le variabili e le costanti di un'istruzione aritmetica devono essere variabili numeriche, tranne le variabili o le costanti appartenenti al tipo stringa di caratteri, al tipo booleano o a qualunque altro tipo che non sia un sottocampo d'interi.
— C'è compatibilità fra il tipo reale ed il tipo intero, nel senso intero — reale: valori interi possono essere usati in espressioni reali, ma il risultato sarà reale. Analogamente, un'espressione intera può essere assegnata ad una variabile reale. Non si può invece assegnare un'espressione reale ad una variabile intera.

5.2.4 - Alcune applicazioni

1. Convertire in Pascal le seguenti espressioni:

   a) $2i + 4j$

   b) $2a + 3b$

   c) $a^2 - 4b$

   d) $2(a + b)$

e) $(a + 2b + 3c)^2$

f) $(\frac{a}{b})^2$

g) $\frac{a + b}{c + d} \times d$

h) $\frac{a}{b} + \frac{b}{d}$

i) $a^2 - 4ab$

l) $\frac{x}{2!} + \frac{x^2}{4!}$

m) a ÷ b (÷ è il simbolo della divisione intera)

n) i modulo (2 × k)

2. Tradurre in Pascal le seguenti espressioni:

$4x + 5y$

$\frac{x + y}{z}$

$n \frac{(n + 1)}{2}$

$(a + b)^2$

$e^{-(x+y)}$

$\log \frac{a + b}{c + d}$

$\sqrt{a^2 + b^2}$

πr^2

$\cos(\Phi)$

3. Date le seguenti espressioni in Pascal, scrivere le espressioni matematiche corrispondenti, e calcolarle per i seguenti valori: a = 2, b = 3, c = 4.

```
a * b + c/a
(a/2 + b/3 + c/4)/3
sqr (a) + sqr (b) + 2 * a * b
sqr (a + b)
a + sqrt (sqr (b) + sqr (c) )/5 * a
c * exp (a * ln (1 + b/100) )
```

4. Date le seguenti istruzioni aritmetiche, calcolare i valori assunti dalle tre variabili a, b e c in seguito all'esecuzione di ogni istruzione, essendo a = 2, b = 3, c = 4.

```
c = a + b + c/a
b = b + c
a = a * b + c * exp (3 * ln (a) )
c = a − b * c
b = (a − 4)/c
a = a − b * c
c = b − a
b = b/2
```

AVVERTENZA

I lettori che dispongono di un sistema che lavora in Pascal possono limitarsi a scrivere le precedenti istruzioni precedute da un'istruzione *leggere* (*a, b, c*), e poi, dopo ogni istruzione, un'istruzione *scrivere* (*a, b, c*).

Soluzione degli esercizi

1.
```
a) 2 * i + 4 * j
b) 2 * a + 3 * b
c) sqr (a) − 4 * b
d) 2 * (a + b)
e) sqr (a + 2 * b + 3 * c)
f) sqr (a/b)
g) (a + b)/(c + d) * d
h) a/b + c/d
i) sqr (a) − 4 * a * b
l) x/2 + sqr (x)/24
m) a DIV b
n) i MOD (2 * k)
```

2.

```
4 * x + 5 * y
(x + y)/z
n * (n + 1)/2
sqr (a + b)
exp (– (x + y))
ln ( (a + b)/(c + d))
sqrt (sqr (a) + sqr (b))
pi * sqr (r)
cos (fi)
```

3.

$ab + \frac{c}{a}$ (valore: 8)

$$\frac{\frac{a}{2} + \frac{b}{3} + \frac{c}{4}}{3}$$ (valore: 1)

$a^2 + b^2 + 2ab$ (valore: 25)

$(a + b)^2$ (valore: 25)

$a + \frac{\sqrt{b^2 + c^2}}{5} \times a$ (valore: 4)

$c\ (1 + \frac{b}{100})^a$ (valore: 4.2436)

4.

	a	b	c
	2	3	4
$c = a + b + c/a$	2	3	7
$b = b + c$	2	10	7
$a = a * b + c * (a)^3$	76	10	7
$c = a - b * c$	76	10	6
$b = \frac{a - 4}{c}$	76	12	6
$a = a - bc$	4	12	6
$c = b - a$	4	12	8
$b = b/2$	4	6	8

5.3 - Le istruzioni di assegnazione di tipo booleano

È possibile definire variabili di tipo booleano ed altresi assegnare a tali variabili il risultato di un'espressione a valore booleano.

5.3.1 - Le costanti booleane o logiche

Sono rappresentate dagl'identificatori standard *true* (*vero*) e *false* (*falso*), e costituiscono a loro volta dei valori assumibili da una qualunque variabile o da una qualunque espressione booleana.

5.3.2 - Gli operatori booleani o logici

Sono tre: l'operatore NOT (NON), l'operatore AND (E) e l'operatore OR (O). In matematica vengono spesso rappresentati in questo modo: NOT con $-$ oppure con $\neg$, AND con $\wedge$ oppure con ., OR con V oppure con +.

5.3.2.1 - L'operatore di negazione logica not (non)

Consideriamo la variabile logica o booleana *a*:

not a è una variabile logica o booleana che è vera quando *a* è falsa, e viceversa.

Questa proprietà si può rappresentare con lo schema seguente, che prende il nome di tabella della verità:

a	not a
vero	falso
falso	vero

Esempio

- Se *a* rappresenta la condizione $b < c$, *not a* esprimerà la condizione inversa: $b \geq c$.
- *Se a* esprime l'affermazione "essere", *not a* esprimerà l'affermazione "non essere".

L'operazione di negazione è detta anche complementazione, e si può indicare con un trattino posto sulla variabile relativa: *a*.

5.3.2.2 - L'operatore or logico (o) o somma booleana

Quest'operatore è detto a volte *or inclusivo*. La suá definizione è la seguente:

Date due variabili booleane *a* e *b*, l'espressione *a or b* è vera se *almeno una* delle due variabili è vera. La tabella della verità è la seguente:

a \ b	vero	falso
vero	vero	vero
falso	vero	falso

a or b

Esempio

Se *a* esprime la condizione "La mia macchina ha un guasto", e *b* esprime la condizionata "Va revisionata", potremo rappresentare la condizione *c* "Bisogna portarla in officina" con l'espressione booleana *c = a or b.*

OSSERVAZIONI

L'*or* booleano non coincide sempre con l' "o" della lingua parlata, che per lo più ha valore esclusivo: "È bel tempo *o* cattivo tempo", "Menù con formaggio *o* dessert".

5.3.2.3 - L'operatore and logico (e) o prodotto booleano

Se consideriamo le due variabili logiche *a* e *b*, l'espressione logica *a and b* è vera solo se *a* e *b* sono *entrambe* vere, come si vede nella seguente tabella della verità:

a \ b	vero	falso
vero	vero	falso
falso	falso	falso

a and b

Esempio

Se *a* esprime la condizione "Piove", e *b* esprime la condizione "Esco", possiamo rappresentare la condizione "Mi bagno" con *c = a and b.*

OSSERVAZIONI

La "e" della lingua parlata è per lo più un *e* logico, eccetto nei casi in cui ha il ruolo di congiunzione.

5.3.3 - Alcune proprietà ed alcuni teoremi dell'algebra di Boole

— *Proprietà commutativa:*

a or b = b or a
a and b = b and a

— *Proprietà associativa:*

a or (b or c) = (a or b) or c
a and (b and c) = (a and b) and c

— *Proprietà distributiva:*

a and (b or c) = (a and b) or (a and c)
a or (b and c) = (a or b) and (a or c)

Quindi gli operatori *and* e *or* hanno, per la proprietà distributiva, ruoli perfettamente simmetrici.

— *Proprietà di complementazione:*
  a or (not a) = true
  a and (not a) = false

— *Involuzione:*
  not (not a) = a

— *Proprietà di trasparenza:*
  a or false = a *false* è elemento trasparente per *or*
  a and true = a *true* è elemento trasparente per *and*

— *Proprietà di "idempotence":*
  a or a = a
  a and a = a

— *Proprietà di assorbimento:*
  a or (a and b) = a
  a and (a or b) = a

— *Teoremi di De Morgan:*
  not (a or b) = (not a) and (not b)
  not (a and b) = (not a) or (not b)

Tutte queste proprietà sono facilmente dimostrabili partendo dalle definizioni degli operatori.

5.3.4 - Le espressioni booleane

Sono espressioni che contengono variabili o costanti booleane separate da operatori booleani.

Esempi

a OR b AND not c
intero OR reale AND not (booleano OR carattere)

Anche le espressioni booleane si valutano tenendo conto di un ordine di precedenza fra gli operatori.
La priorità più alta è accordata all'operatore NOT, che è un operatore *monadico*, perché interviene su un solo operando.
Gli operatori AND e OR sono operatori *diadici*, perché agiscono su due operandi.

Fra i due operatori la priorità spetta all'operatore AND (prodotto logico).

L'operatore OR (somma logica) ha così la priorità più bassa.

Anche qui si possono usare espressioni poste fra parentesi, che verranno valutate per prime.

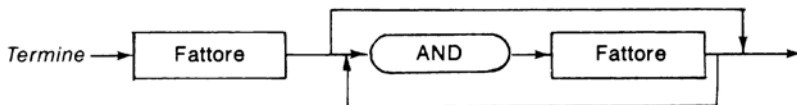
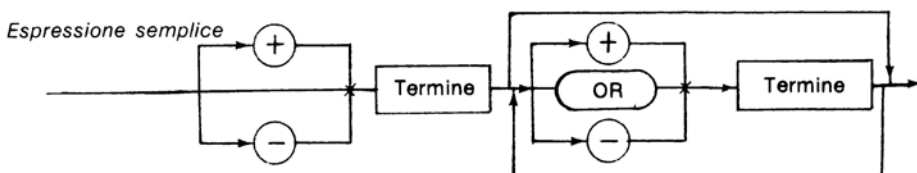
Ad esempio, l'espressione *a OR b AND c* è valutata eseguendo prima *b AND c*, e poi *a OR (b AND c)*.

Volendo eseguire per prima l'operazione OR, si sarebbero dovute usare delle parentesi: (*a OR b*) *AND c*.

Analogamente l'espressione *a AND b OR NOT c AND d* è valutata così: (*a AND b*) *OR* (*NOT* (*c*) *AND b*).

Dal punto di vista sintattico, avremo i seguenti diagrammi:

– L'operatore NOT (NON) viene introdotto a livello di fattore:

– L'operatore AND (E) è introdotto a livello di termine:

– Infine, l'operatore OR (O) è introdotto a livello di espressione semplice:

Qui abbiamo rappresentato il diagramma *completo* di un'espressione semplice, ma si tenga presente che anche i diagrammi dei fattori e dei termini possono venir rappresentati nello stesso modo, comprendendovi anche tutti gli operatori aritmetici già visti. Troverete questi diagrammi nell'Appendice 2.

Da tutto questo appare che, dal punto di vista sintattico, il concetto di espressione semplice si applica altrettanto bene alle espressioni aritmetiche e alle espressioni booleane.

Il problema della validità di queste espressioni, tenuto conto delle dichiarazioni relative, è un problema semantico; quindi, se la variabile che compare al primo membro è booleana, l'espressione contenuta nel secondo membro dev'essere valutata secondo la logica booleana.

Questo comunque non vuol dire che tutti i termini o i fattori dell'espressione debba-

no essere booleani. Infatti vedremo che le espressioni in Pascal hanno in sè il concetto di relazione: queste espressioni possono quindi essere valutate anche secondo la logica booleana.

5.3.5 - Le funzioni booleane standard

Così come esistono funzioni matematiche standard, allo stesso modo esistono alcune funzioni booleane standard, dette anche *predicati.*

5.3.5.1 - La funzione odd

Questa funzione ha come parametro un valore intero: *odd* (*x*) con *x* di tipo intero. Il risultato della funzione è *vero* se il valore di *x* è *dispari*, *falso* nel caso contrario.

5.3.5.2 - La funzione eoln

Questa funzione ha come parametro il nome di un flusso, che può essere omesso se il flusso è quello standard *input* (*ingresso*). Il risultato della funzione è *vero* quando il carattere che viene letto in corrispondenza di un'operazione d'ingresso dati è un carattere di fine linea (ritorno a capo); il risultato è *falso* per tutti gli altri caratteri.

5.3.5.3 - La funzione eof

Anche questa funzione ha come parametro un nome di flusso, implicito nel caso del flusso standard *input* (*ingresso*).

Il risultato della funzione è *vero* quando si trova un carattere di fine flusso (*eof*: end of file = fine del flusso).

Esempio

```
PROGRAM fbool;
VAR i: integer; dispari: boolean;
     finelinea: boolean;
BEGIN
  read (i); dispari: = odd(i); finelinea: = eoln;
  IF NOT finelinea THEN
  IF dispari THEN write (i,'dispari')
    ELSE write (i,'pari');
END.
```

5.3.6 - Le espressioni relazionali

Un'espressione relazionale permette di mettere in relazione due espressioni semplici: essa è espressa mediante gli operatori relazionali che elenchiamo qui di seguito. Un'espressione relazionale che pertanto dà come risultato un valore booleano (*vero* o *falso*).

5.3.6.1 - Gli operatori relazionali

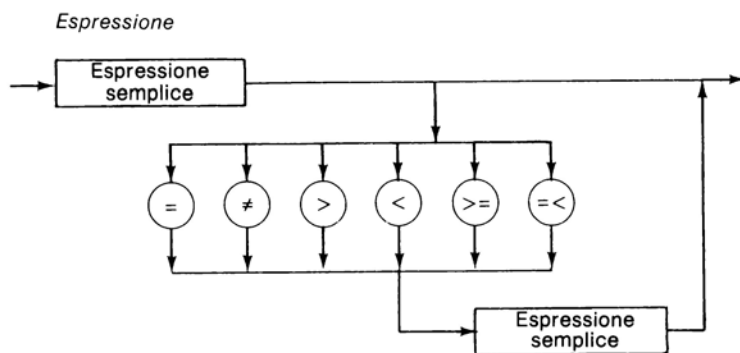
Gli operatori relazionali in Pascal sono sei:

=	uguaglianza
< >	diverso (≠)
>	maggiore
<	minore
>=	maggiore o uguale (≥)
<=	minore o uguale (≤)

Esempi

a < > b *a* diverso da *b*
c > = 10 *c* maggiore di o uguale a 10
a < b + c *a* minore di *b* + *c*
a = b + c *a* uguale a *b* + *c*

5.3.6.2 - Sintassi delle espressioni relazionali

La rappresentazione sintattica delle espressioni relazionali è data dal diagramma seguente:

Si è visto che nelle espressioni semplici rientrano le espressioni booleane e le e-spressioni in cui compaiono i tipi carattere e i tipi non standard: quindi in Pascal il concetto di espressione relazionale è estremamente generale. L'espressione relazionale viene detta semplicemente *espressione*, perché comprende in sé tutti gli altri tipi di espressione definibili in Pascal.

Ogni espressione può essere assegnata ad una variabile dello stesso tipo, come si è già visto per le espressioni logiche. Nella misura in cui le espressioni relazionali sono valutate secondo la logica booleana, possono venir assegnate a variabili booleane.

Esempio

```
VAR x, y, z: real;
     uguale, magg, min: boolean;
BEGIN
  uguale: = x = y; (*uguale sarà vero se x = y*)
  magg: = y > z; (*magg sarà vero se y > z*)
  min: = x < y - z; (*min sarà vero se x < y - z*)
END.
```

5.4 - Generalizzazione dell'istruzione di assegnazione

Il concetto di espressione è stato generalizzato per potervi comprendere le espressioni relazionali. Possiamo quindi definire un'istruzione di assegnazione generalizzata in questo modo:

```
variabile: = espressione;
```

Il concetto di espressione è stato definito dal punto di vista sintattico in una forma tale da permettere di valutare un'espressione qualunque in modo non ambiguo. Un'istruzione di questo tipo può pertanto essere sintatticamente corretta, e pure esser priva di senso per quanto concerne il tipo della variabile e gli elementi contenuti nell'espressione.

L'istruzione di assegnazione dunque è valida per tutti i tipi di variabili, compresi i tipi carattere, stringa di caratteri ed i tipi non standard, a patto che l'espressione possa venir valutata all'interno del medesimo tipo.

Esempio

```
TYPE grandezza = (piccolo, medio, grande);
      durata = (immediato, breve, abbastanzalungo, lungo);
VAR misura: grandezza;
     tempo: durata; car: char;
BEGIN
  read (car);
  IF car = 'g' THEN misura: = grande;
     ..........
  IF car = 'b' THEN tempo: = breve;
     ..........
END.
```

5.4.1 - Generalizzazione dell'impiego degli operatori relazionali

Solitamente gli operatori relazionali sono riservati al confronto fra espressioni arit-

metiche. In Pascal sono utilizzabili qualunque sia il tipo di variabile scalare adottato, a patto però che le due espressioni messe a confronto siano dello stesso tipo.

In linea generale supporremo dunque che esista una relazione d'ordine per l'insieme degli elementi costituenti un tipo, standard o non standard; esiste poi anche una relazione d'intensità.

5.4.1.1 - Il tipo carattere

La relazione d'ordine è definita dall'ordine nel quale la sequenza di caratteri è stata disposta, nel codice associato o nel tipo relativo.

La relazione d'identità è definita dall'identità dei due caratteri.

Dal momento che i codici sono diversi a seconda dei sistemi e dei calcolatori usati, la funzione *ord*(*car*) permetterà di conoscere l'ordine esatto adottato nel codice associato ai caratteri. Sapendo che in tutti i calcolatori il codice rispetta l'ordine alfabetico, si ha:

'a' 'b' 'c' .... 'x' 'y' 'z'

Analogamente, per i caratteri decimali si ha:

'0' '1' '2' '3' .... '8' '9'

Per gli altri caratteri è necessario verificare il codice impiegato dal sistema sul quale si lavora.

Esempio

```
VAR car1, car3, car2: char;
```

Si possono poi avere istruzioni del tipo

```
car1: = 'e'; car2: = 't'; car3: = '4';
```

In questo caso abbiamo $car1 < car2$.

In codice ASCII $car3 < car1$, ma in un altro codice potrebbe non essere cosi.

AVVERTENZA

Il risultato che si ottiene applicando *ord* ad un carattere cifra non è uguale al valore della cifra stessa. Ad esempio, $ord(car3) \neq 4$.

Invece $ord(car3) - ord('0') = 4$, e, analogamente, $ord(car1) - ord('a') = 4$. Infatti, essendo *car1* uguale al carattere *e*, la differenza fra il codice di *a* ed il codice di *e* è 4.

5.4.1.2 - Il tipo booleano

In questo caso occorre stabilire una relazione d'ordine fra i due valori booleani *true*

(*vero*) e *false* (*falso*). In Pascal standard abbiamo: *false* (*falso*) < *true* (*vero*). Questo si giustifica col fatto che il valore logico *falso* è tipicamente rappresentato dal valore 0, il valore *vero* dal valore 1.

L'identità di due espressioni logiche è data dall'uguaglianza dei valori logici delle due espressioni.

Esempio

```
VAR a, b, c: boolean;
```

Si possono definire le seguenti istruzioni:

```
a: = 5 < 2;
b: = a OR (6 < 8);
c: = a AND b;
```

a è sempre falso, e *b* è sempre vero, per cui anche *c* è falso. Abbiamo pertanto l'identità

a = c > b   (*c* è falso, *b* è vero, per cui *c* > *b* è falso)

5.4.1.3 - I tipi non standard

La relazione d'ordine che interviene in questo caso è data dal numero d'ordine all'interno della sequenza che definisce il tipo non standard. L'uguaglianza è data dall'identità di due valori del medesimo tipo.

Il numero d'ordine del primo valore definito in un tipo scalare è zero (0).

Esempio

```
TYPE misura = (piccolo, medio, grande);
     sesso = (maschile, femminile);
     mese = (gennaio, febbraio, ..., dicembre);
```

Si ha:

piccolo < medio < grande

Se la definizione fosse stata:

misura = (grande, medio, piccolo);

avremmo avuto:

grande < medio < piccolo

Analogamente, definendo una variabile *persona*:

```
VAR persona: sesso;
    uomo, donna: boolean;
```

abbiamo:

```
uomo: = persona = maschile;
donna: = persona > maschile;
```

Infine, essendo *ord* (*gennaio*) uguale a 0, e *ord* (*dicembre*) uguale a 11, abbiamo:

```
gennaio < febbraio < ... < novembre < dicembre
```

5.4.1.4 - Il tipo stringa di caratteri (string)

Anche se si tratta di un tipo non standard, è utile studiarne le espressioni relazionali. Il tipo stringa di caratteri è disponibile sul Pascal dell'U.C.S.D.

L'ugualianza di due stringhe comporta l'identità di due sequenze di caratteri, compresi i caratteri bianchi.

La relazione d'ordine è definita dall'ordine lessicografico associato al codice associato (il codice ASCII). Il confronto di due stringhe permette, in particolare, di ordinare i caratteri alfabetici nell'ordine naturale. Su questo torneremo in un successivo capitolo.

Esempio

```
PROGRAM confrontastringa;
VAR c, testo: string; conf: boolean;
BEGIN
   testo: 'blablabla';
   c: = 'blabla';
   conf: = c < testo; (*è vero*);
   conf: = c = 'bla bla'; (*è falso*)
   testo: = 'Brando';
   conf: = c < testo; (*è vero*)
   c: = 'Marlon Brando';
   conf: = c > testo; (*è vero*)
   testo: = 'Brando Marlon';
   conf: = c < > testo; (*è vero*)
END.
```

5.4.1.5 - Esempi di assegnazione booleana

1)

```
PROGRAM Boole;
VAR a, b, c: boolean;
BEGIN
   a: = true; b: = false;
   c: = a OR b;
   a: = NOT (b AND c) OR b;
END.
```

2)
```
PROGRAM relazione;
VAR eta, altezza: integer;
     maggiorenne, alto, normale, basso: boolean;
BEGIN
  maggiorenne: = eta > 18;
  alto: = altezza > 180;
  basso: = altezza < 150;
  normale: = (NOT alto) AND (NOT basso);
  (*oppure*)
  normale: = (altezza ≥ 150) AND (altezza ≤ 180);
END.
```

3)
```
PROGRAM domandarisposta;
VAR risposta, domanda: string; numero: integer;
     Maria, figlio, esente: boolean;
BEGIN
  domanda: = 'siete Maria?';
  write (domanda); read (risposta);
  Maria: = risposta = 'sì';
  domanda: = 'quanti figli avete?';
  write (domanda); read (numero);
  figlio: = numero > 0;
  esente: = Maria AND figlio;
END.
```

5.5 - Le istruzioni composte

Abbiamo visto che in Pascal il punto e virgola agisce da separatore fra le istruzioni elementari.Si possono anche definire delle istruzioni composte, per mezzo di blocchi di istruzioni elementari delimitati dalle parole BEGIN e END (INIZIO e FINE).

Esempio

```
INIZIO
  leggere (dato);
  totale: = totale + dato
FINE;
```

In un'istruzione composta tutte le istruzioni elementari devono essere separate da punti e virgola, tranne l'ultima, che precede la parola riservata END (FINE).

In un programma si possono annidare l'una nell'altra più istruzioni composte; quindi può essere utile precisare il livello della struttura. Per far questo si dovrà ricorrere ad un accorgimento: lo sfasamento del margine sinistro e l'aggiunta di commenti.

Esempio

INIZIO (*istruzione di livello 1*)

....

INIZIO (*istruzione di livello 2*)

....

FINE (*istruzione di livello 2*)

....

FINE (*istruzione di livello 1*)

5.6 - La struttura di selezione

5.6.1 - La struttura di selezione semplice

Questa struttura è detta anche *test*, perché permette di verificare una condizione espressa in forma booleana. A seconda che la valutazione sia *vera* o *falsa*, si dovrà eseguire un'elaborazione differente.

Questo meccanismo di selezione si può esprimere nella sua forma più generale con un'espressione del tipo

se condizione *allora* f *sennò* g

Se la condizione è vera, si esegue l'elaborazione *f*; se la condizione è falsa, si esegue l'elaborazione *g*.

Esempi

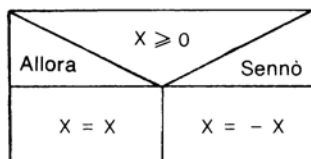
1) La funzione valore assoluto è definita da

$|x| = x$ *se* $x \geq 0$

$|x| = -x$ *se* $x < 0$

Questo si può esprimere come segue:

SE x ≥ 0 ALLORA x: = SENNÒ x: = –x

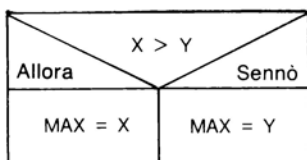
Il grafo GNS corrispondente è:

Considerando la condizione inversa, a questa struttura si sostituirà la seguente:

SE x < 0 ALLORA x: = –x

il che equivale a considerare l'opposto di *x*, nel caso in cui il suo valore sia negativo.

2) La funzione massimo fra due numeri sarà definita da
   max (x, y) = x *se* $x \geq y$
   max (x, y) = y *se* $x < y$

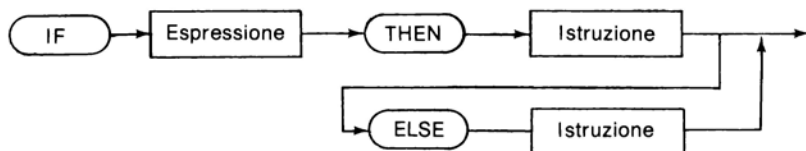
ovvero
   se $x > y$ *allora* max (x, y) = x *sennò* max (x, y) = y

Allora il grafo GNS corrispondente è:

Questo si può programmare in due modi:
   se x > y ALLORA max: = x SENNÒ max: = y

oppure
   max: = x; SE x < y ALLORA max: = y

Da questi due esempi elementari si vede che per una struttura di selezione semplice sono possibili due forme: quella più generale fa intervenire i due termini dell'alternativa SE ... ALLORA ... SENNÒ; l'altra presuppone che uno dei termini (SENNÒ) corrisponda alla sequenza dell'algoritmo, o del programma.

In Pascal la struttura dell'istruzione di selezione consente di usare tanto l'una quanto l'altra forma.

Il grafo sintattico è pertanto il seguente:

Nella versione inglese la struttura è:

   IF espressione THEN istruzione ELSE istruzione

Questo presuppone che l'espressione che vien dopo il SE sia valutabile secondo la logica booleana: dovrà quindi essere o un'espressione booleana pura, o un'espressione relazionale.

Dal punto di vista sintattico sono possibili entrambe le forme, con o senza la clausola SENNÒ. La struttura di selezione non vuole il punto e virgola, a meno che ci siano delle istruzioni composte.

Esempi

1)
```
PROGRAM test;
VAR risp: char;
BEGIN
  write ('fa bel tempo?');
  read (risp);
  IF risp = 's' THEN writeln ('esco')
   ELSE writeln ('vado al cinema');
END.
```

In questo programma viene posta la domanda. La risposta *s* (*sì*) provoca la scrittura dei messaggio 'esco'. Qualunque altra risposta (interpretata come negazione di 'fa bel tempo') porterà alla scrittura del messaggio 'vado al cinema'.

Supponiamo ora di aver scritto:

```
IF risp = 'n' THEN
writeln ('prende l''ombrello');
writeln ('esce');
```

In questo caso, se la risposta è *n* (*no*), viene scritto successivamente 'prende l'ombrello' ed 'esce'; per qualunque altra risposta si scrive soltanto 'esce'.

Quando si usa una struttura di selezione bisogna stare attenti a ricorrere alla seconda forma solo se l'elaborazione corrispondente alla clausola ALLORA (condizione vera) dev'essere seguita dall'elaborazione che viene eseguita nel caso in cui la condizione è falsa.

2) Si debba calcolare la radice quadrata di $|x|$ non ricorrendo alla funzione valore assoluto:

```
PROGRAM radass;
VAR x: real;
BEGIN
  read (x);
  IF x < 0 THEN x: = -x;
  writeln (sqrt (x) );
END.
```

Se x è negativa, si considera l'opposto di x e si scrive la radice quadrata (*sqrt*); se x è positiva, si scrive direttamente il risultato.

3) Invece, se nell'esempio del massimo fra due numeri scriviamo:

```
PROGRAM max;
VAR x, y: real;
BEGIN
  read (x, y);
  IF x < y THEN max: = x;
  max: = y;
  writeln (max);
END.
```

otteniamo un programma non corretto, perché il risultato è in ogni caso
max = y

Di conseguenza bisogna scrivere:

```
BEGIN
  read (x, y);
  max: = x;
  IF x < y THEN max: = y;
  writeln (max);
END.
```

In questo caso è preferibile utilizzare la struttura di selezione completa:

```
BEGIN
  read (x, y);
  IF x > y THEN max: = x ELSE max: = y;
  writeln (max);
END.
```

5.6.2 - Uso delle istruzioni composte

L'utilità della struttura di selezione, in Pascal, sta nel fatto che permette di usare istruzioni composte nei due termini in alternativa. Si hanno cosi strutture della forma

```
SE espressione
    ALLORA INIZIO
              istruzione 1;
              istruzione 2;
              ..........
              istruzione n
              FINE
```

```
SENNÒ INIZIO
        istruzione 1;
        ..........
        istruzione m
        FINE;
```

Anche qui l'alternativa SENNÒ può non esserci. Questa struttura permette, in particolare, d'inserire in uno dei due termini in alternativa istruzioni d'iterazione (v. oltre), o altre istruzioni di selezione.

In questo modo si ottengono delle istruzioni di selezione composta:

```
SE condizione 1
    ALLORA SE condizione 2
       ALLORA SE condizione 3
          ALLORA istruzione 3
          SENNÒ istruzione 3n
       SENNÒ istruzione 2n
    SENNÒ istruzione 1n
```

Dall'esame del diagramma sintattico si vede, da una parte, che non può esserci punto e virgola immediatamente prima della clausola SENNÒ, e, dall'altra, che l'assenza della clausola SENNÒ viene interpretata in modo non ambiguo associando questa clausola alla selezione più interna (il SE più vicino) che non le è stata ancora associata.

Consideriamo, ad esempio, la seguente struttura:

```
SE condizione 1
    ALLORA SE condizione 2
              ALLORA istruzione 2
              SENNÒ istruzione 2n
```

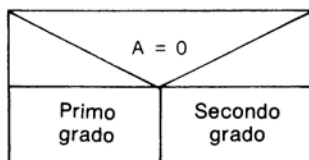
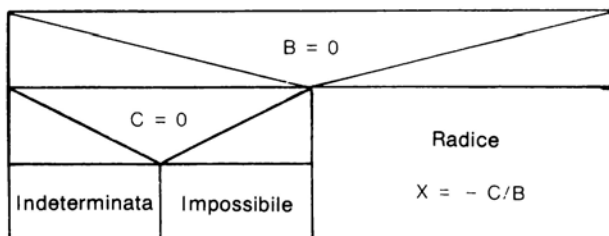
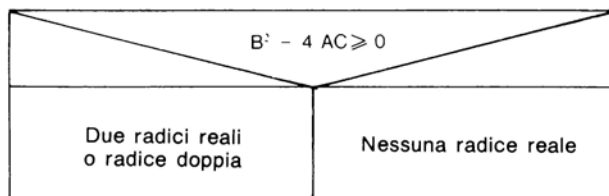
In questo caso il SENNÒ si riferisce alla seconda struttura di selezione (la condizione 2). La prima selezione non richiede questo tipo di clausola.

È invece ovvio che si avranno tante clausole ALLORA quante sono le clausole di selezione: cioé tanti SE ed altrettanti ALLORA. In particolare, una clausola SENNÒ non può venire immediatamente dopo una condizione: in altre parole, un SENNÒ ed un SE devono sempre essere separati da un ALLORA.

5.6.3 - Un esempio applicativo

Si debba scrivere un programma che fornisca le radici reali di un'equazione di secondo grado

$$ax^2 + bx + c = 0$$

Sappiamo che bisogna innanzitutto verificare se *a* è o no nullo. Il diagramma GNS sarà dunque il seguente:

Nel caso *primo grado*, bisogna poi verificare se *b* e/o *c* sono nulli. Cioè:

Nel caso *secondo grado*, il discriminante $b^2 - 4ac$ individua il caso in cui esistono radici reali:

Si ottiene allora il seguente grafo GNS:

A = 0

Allora
primo grado

Sennò
secondo grado

B 0

Allora
soluzione
degenerata

Sennò
una radice

Delta =
$B^2 - 4AC \geqslant 0$

Allora
soluzioni

Sennò
nessuna
soluzione

C = 0

Allora

Sennò

Scrivere

indeter-
minato

Scrivere

impos-
sibile

X --C/A

Scrivere X

Delta $\neq$ 0

Allora

Sennò

X1 =

$\frac{-B-\sqrt{Delta}}{2A}$

X2 =

$\frac{-B+\sqrt{Delta}}{2A}$

Scrivere
X1, X2

X =
-- B/2A

Scrivere
radice
doppia
X

Scrivere

Nessuna
radice
reale

Sulla base di questo grafo è facile procedere alla programmazione. Nel programma che segue non è contemplato il caso della radice doppia. Il lettore può aggiungerlo, a titolo di esercizio, e può aggiungere altresì, nel caso in cui il discriminante sia negativo, il calcolo delle parti reali e immaginarie.

```
PROGRAM secondogrado;
uses transcend;
VAR a, b, c: real;
    delta: real;
    x1, x2: real;
BEGIN
  writeln ('introdurre i coefficienti');
  read (a, b, c);
  IF a = 0 THEN
```

```
        BEGIN
        IF b = 0 THEN IF c = 0 THEN writeln ('radici indeterminate')
                                ELSE writeln ('radici impossibili')
                 ELSE writeln ('radice di primo grado =', - c/b)
        END
        ELSE
          BEGIN
          delta: = b * b - 4 * a * c;
          IF delta < 0 THEN writeln ('nessuna radice reale')
          ELSE
            BEGIN
            x1: = (-b + sqrt (delta) ) / (2 * a);
            x2: = (-b - sqrt (delta) ) / 2 / a;
            writeln ('x1 =', x1, 'x2 =', x2)
            END
          END;
    END.
```

Qui di seguito sono riportati alcuni esempi di esecuzioni:

```
0    0    2                                    equazione 2 = 0
radici impossibili

0    0    0                                    equazione 0 = 0
radici indeterminate

1    1    1                                    equazione x² + x + 1 = 0
nessuna radice reale

0    2    3                                    equazione 2x + 3 = 0
radice di primo grado

1    -1    -2                                  equazione x² - x - 2 = 0
x1 = 2.000000   x2 = -1.000000

1    -6    9                                   equazione x² - 6x + 9 = 0
x1 = 3.000000   x2 = 3.000000
```

OSSERVAZIONI

a) Nel programma abbiamo usato la clausola *uses transcend*, che è particolare del Pascal U.C.S.D., e indica la necessità di utilizzare le funzioni matematiche (trascendenti): questo è reso necessario dalla funzione radice quadrata *(sqrt)*.
b) Si è visto che in Pascal le istruzioni di assegnazione si possono utilizzare con espressioni booleane o relazionali.

Nel programma avremmo potuto definire delle variabili booleane, *grado1* e *solreale*, in questo modo:

```
grado1: = a = 0;
solreale: = b * b - 4 * a * c ≥ 0
```

In tal caso avremmo avuto una struttura del tipo:

```
SE grado1 ALLORA (*elaborazione di primo grado*)
                 ..........
          SENNÒ
             SE solreale ALLORA
                (*elaborazione di secondo grado*)
                ..........
             SENNÒ
                (*elaborazione del caso complesso*)
```

5.6.4 - L'istruzione di selezione multipla

L'istruzione di selezione semplice permette di scegliere fra i due termini di un'alternativa. In alcuni casi la selezione dev'essere operata entro un numero di possibilità maggiore di due. Per quanto una tale situazione possa venir sempre scomposta in una serie di selezioni semplici, si può utilizzare più efficacemente l'istruzione di selezione multipla, che è una generalizzazione della struttura di selezione semplice.

Si abbia ad esempio il caso in cui si voglia conoscere la riduzione applicabile ad un utente delle ferrovie, che potrà essere un viaggiatore ordinario (tariffa intera), un abbonato (riduzione del 50%), un bambino di meno di cinque anni (biglietto gratuito), un pensionato (riduzione del 50%), un membro di una famiglia numerosa, etc.

La dichiarazione sia la seguente:

```
TYPE viaggiatore = (bambino, abbonato, pensionato, ordinario,
     trenta quaranta, cinquanta);
VAR  persona: viaggiatore;
     riduz: real;
```

Con una struttura di selezione semplice bisognerebbe verificare tutti i casi possibili:

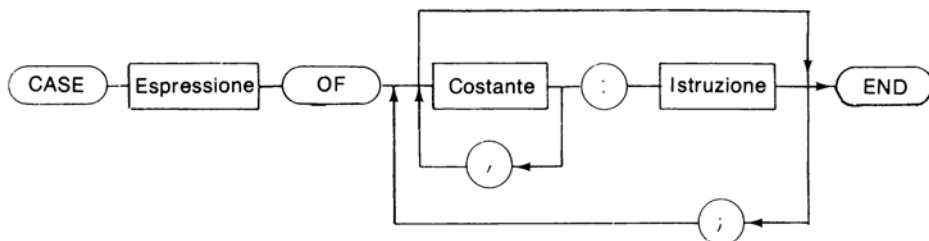
```
SE persona = bambino ALLORA riduz: = 100
   SENNÒ SE persona = abbonato ALLORA riduz: = 50
      SENNÒ SE persona = pensionato
               ALLORA riduz: = 50
         SENNÒ SE persona = ordinario
                  ALLORA riduz: = 0
```

La struttura di selezione multipla evita questi incastri di selezioni semplici, permettendo di specificare i vari casi senza dover scrivere le strutture SE ALLORA SENNÒ.

Quest'istruzione prende il nome di struttura CASE (CASO) ... OF (FRA), nome che indica appunto la selezione di un *caso fra* un numero di possibilità definito nella struttura di selezione multipla. Così, nell'esempio precedente si scriverà:

```
CASE persona OF          (*caso persona fra*)
  bambino: riduz: = 100;
  abbonato: riduz: = 50;
  pensionato: riduz: = 50;
  ordinario: riduz: = 0;
  trenta: riduz: = 30;
  quaranta: riduz: = 40;
  cinquanta: riduz: = 50
END
```

Una struttura di questo tipo è molto più chiara e concisa della precedente.

Il diagramma sintattico è il seguente:

In questa struttura l'espressione che vien dopo la parola riservata CASE prende anche il nome di *selettore*. Il tipo del selettore è indifferente, a parte il fatto che deve corrispondere al tipo delle costanti che identificano ciascuna scelta possibile nella struttura di selezione multipla. Aggiungiamo che il tipo del selettore dev'essere scalare, ma *non reale*.

Rappresentiamo questa struttura come segue:

```
CASO  selettore FRA
        selezione 1: istruzione 1;
        selezione 2: istruzione 2;
        selezione 3: istruzione 3;
          ..........
        selezione k: istruzione k
FINE
```

Questa struttura è detta anche *griglia di selezione.*

Durante l'esecuzione il valore del selettore deve corrispondere ad una delle possibili selezioni; viene allora eseguita l'istruzione associata, dopo che il controllo è trasferito all'istruzione che vien dopo la fine della struttura. Se il valore del selettore non corrisponde ad alcuna delle selezioni elencate, il risultato è indeterminato, e diverso a seconda dei sistemi.

È chiaro che questa struttura presuppone che le scelte, o selezioni, si escludano a vicenda.

Quando a più scelte (o selezioni) corrisponde una medesima elaborazione, si può raggrupparle, separandole per mezzo di virgole.

Esempi

1) Nell'esempio delle tariffe ferroviarie varie categorie avevano diritto alla riduzione del 50%: possiamo allora raggrupparle come segue:

```
CASE persona OF
        bambino: riduz: = 100;
        abbonato, pensionato, cinquanta: riduz: = 50;
        ordinario: riduz: = 0;
        trenta: riduz: = 30;
        quaranta: riduz: = 40
END
```

Le costanti che identificano ciascuna selezione sono dette a volte etichette di CASE, ma questo non è legittimo, perché gl'identificatori, o costanti, non devono essere menzionati nella dichiarazione delle etichette, e inoltre non possono essere inseriti in altre istruzioni.

2) In tutti i casi il selettore può essere di tipo intero: basta infatti mettere al suo posto la funzione *ord* (*selettore*), e le selezioni saranno sostituite da numeri interi.

Allora l'esempio precedente diventerà:

```
CASE ord (persona) OF
             0: riduz: = 100;
             1, 2, 6: riduz: = 50;
             3: riduz: = 0;
             4: riduz: = 30;
             5: riduz: = 40
END
```

3) Per esprimere il selettore si possono usare anche delle variabili di tipo carattere. Sempre nell'esempio precedente, supponiamo di aver definito un codice che rap-

presenti le varie categorie di persone: *b* per bambino, *a* per abbonato, *p* per pensionato, e cosi via. Avremo allora:

```
CASE codice OF
   'b': riduz: = 100;
   'a', 'p', 'c': riduz: = 50;
   'o': riduz: = 0;
   't': riduz: = 30;
   'q': riduz: = 40
END
```

4) Alla struttura di selezione semplice si può sostituire una struttura che usa l'istruzione CASO. Così, ad esempio, la struttura

```
SE condizione ALLORA istruzione 1
              SENNÒ istruzione 2
```

è equivalente alla struttura

```
CASO condizione FRA
   vero: istruzione 1;
   falso: istruzione 2
FINE
```

ed anche alla struttura

```
CASO ord (espressione booleana) FRA
   0: istruzione 2;
   1: istruzione 1
FINE
```

5.7 - Le strutture iterative in Pascal

Le istruzioni iterative costituiscono l'ultimo gruppo di istruzioni indispensabili nella programmazione di algoritmi.

Una struttura iterativa, o ripetitiva, permette di specificare che un gruppo d'istruzioni sarà eseguito un certo numero di volte.

Il numero delle iterazioni, o ripetizioni, è fissato mediante un criterio di arresto caratteristico della struttura utilizzata.

Abbiamo visto che in Pascal esistono tre strutture iterative: si può richiedere l'esecuzione di un blocco di istruzioni *finquando* una certa condizione è verificata; oppure si può specificare la ripetizione di un blocco *finché* una data condizione si realizza; o ancora si può richiedere l'esecuzione di un ben preciso numero d'iterazioni.

Tutte queste strutture costituiscono quello che chiamiamo ciclo di programma. Dato un algoritmo, è sempre possibile ricorrere ad una qualunque di esse, ma generalmente ce n'è una che è più appropriata delle altre.

Quindi, prima di procedere alla programmazione, bisogna scegliere oculatamente la struttura più adatta.

Presenteremo adesso le tre strutture dal punto di vista sintattico, riferendoci ad esempi semplici, al fine di evidenziare il passaggio da una forma all'altra ed i vantaggi propri di ciascuna struttura.

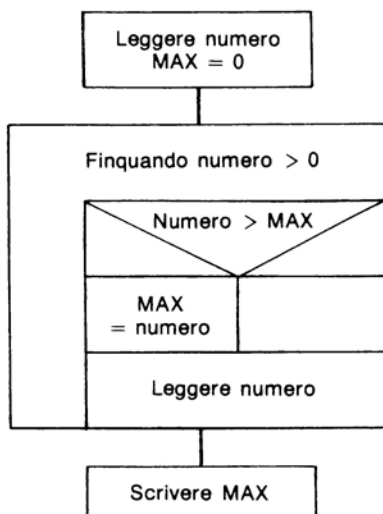
5.7.1 - L'istruzione iterativa while (finquando)

Supponiamo ad esempio di dover scrivere un programma di ricerca del valore massimo all'interno di una sequenza di numeri positivi, essendo nullo o negativo l'ultimo valore della sequenza.

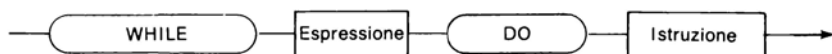
L'algoritmo corrispondente presuppone inoltre che i dati vengano letti ad uno ad uno, e che i dati letti vengano poi confrontati con il valore massimo, che viene quindi memorizzato.

L'algoritmo si può esprimere come segue:

```
leggere numero
sia max = 0
finquando numero > 0 fare:
   se numero > max allora max = numero
   leggere numero
scrivere massimo
fine
```

Il grafo GNS è:

Il programma corrispondente è:

```
PROGRAM massimo;
VAR max, numero;
BEGIN
  read (numero); max: = 0;
  WHILE numero > 0 DO
    BEGIN
      IF numero > max THEN max: = numero;
      read (numero)
    END;
  writeln ('massimo =', max);
END.
```

La sintassi dell'istruzione WHILE (FINQUANDO) è definita dal seguente diagramma:

L'espressione dev'essere valutabile in forma booleana: pertanto dovrà essere o un'espressione logica, o un'espressione relazionale esprimente una condizione.

La semantica dell'espressione esprime la ripetizione dell'istruzione posta dopo il DO (FARE), *finquando* la condizione espressa dall'espressione è *vera.*

In particolare, se l'espressione è falsa, non ci sarà iterazione; se invece l'espressione è sempre vera, il numero delle iterazioni effettuate sarà infinito.

Si tenga presente che l'istruzione che vien dopo il DO (FARE) può essere un'istruzione composta.

Perché una struttura iterativa *finquando* abbia termine, occorre che l'espressione che traduce la condizione di arresto venga modificata nell'istruzione che vien dopo il DO (FARE). Ma questa è una condizione necessaria, ma non sufficiente: infatti, in linea generale, anche se il ciclo termina, può darsi tuttavia che esistano dei valori specifici delle variabili che intervengano nella condizione, che fanno sì che il ciclo sia infinito. Consideriamo, ad esempio, le istruzioni:

```
FINQUANDO NON dispari (numero) FARE
  numero: = numero DIV 2;
```

L'istruzione termina in tutti i casi, tranne quando *numero* è nullo, nel qual caso occorre aggiungere un test prima della struttura iterativa, per evitare di ritrovarsi con un ciclo infinito.

Una situazione di questo tipo è possibile anche se si utilizza un predicato come fine di linea (*eoln*) o fine di flusso (*eof*). Infatti, se scriviamo

```
WHILE NOT eof DO
```

e, per errore, nel flusso su cui si lavora (o nelle istruzioni che seguono) non si trova la fine del flusso, il ciclo sarà infinito, a meno di prevedere di uscire dal ciclo per errore non appena s'incontra un dato non corrispondente ai valori o ai tipi previsti.

È altrettanto importante sottolineare che bisogna far bene attenzione al numero esatto d'iterazioni effettuate. La sequenza d'istruzioni

```
FINQUANDO nriterazione ≤ n FARE
          nriterazione: = nriterazione + 1;
          scrivere (nriterazione, '=', n + 1);
```

darà effettivamente un numero d'iterazioni uguale ad $n + 1$, perché, per rendere falsa la condizione *nriterazione*, bisogna che il numero d'iterazioni sia uguale ad $n + 1$.

5.7.1.1 - Due algoritmi di calcolo del M.C.D.

Il massimo comun divisore (M.C.D.) di due numeri si può calcolare con più di un algoritmo:

1) Un primo algoritmo consiste nel detrarre successivamente a da b, o b da a, a seconda che $a > b$ o no, fino ad avere $a = b$, condizione che indica che il M.C.D. è stato trovato.

Questo algoritmo suppone noto il fatto che

$$MCD\ (a, b) = MCD\ (a - b, b) \qquad \text{per } a > b$$

che si può esprimere con una struttura del tipo:

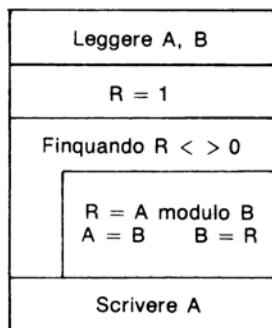
```
FINQUANDO a < > b FARE
   SE a > b ALLORA a: = a - b
            SENNÒ b: = b - a
```

Avremo allora il programma seguente:

```
PROGRAM mcd2;
VAR a, b: integer;
BEGIN
  writeln ('introdurre due numeri');
  read (a, b);
  (*calcolo del mcd*)
  WHILE a < > b DO
    IF a > b THEN a: = a − b
      ELSE b: = b − a;
  write (a);
END.
```

2) Il secondo algoritmo fa intervenire il resto della divisione intera di *a* per *b*:

siano q = a ÷ b ed r = a − b * q (oppure r = x a modulo b)
se r è nullo, *allora* il M.C.D. è b
sennò, si sostituisce a con b e b con r, e si compie un'altra iterazione.

Il grafo GNS è il seguente:

Il programma è pertanto:

```
PROGRAM mcd;
VAR a, b, q, r: integer;
BEGIN
  writeln ('introdurre 2 interi');
  read (a, b);
  write ('mcd di', a, 'e', b, '=');
  r: = 1;
  WHILE r < > 0 DO
```

```
      BEGIN
        q: = a DIV b;         )
        r: = a - b * q;       } oppure r: = a MOD b
        a: = b; b: =r;        )
      END;
    write (a);
  END.
```

Si noti che, perché l'algoritmo sia corretto, bisogna inizializzare *r* ad un valore diverso da zero. Vedremo che per questo caso è più appropriata la struttura iterativa ripetere.

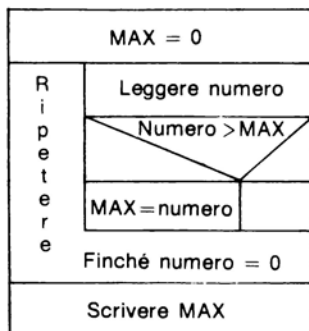
5.7.2 - La struttura iterativa ripetere finché

La seconda struttura iterativa disponibile in Pascal è la struttura REPEAT (RIPETERE) ... UNTIL (FINCHÈ).

Riprendiamo l'algoritmo di ricerca del numero massimo di una sequenza di numeri positivi, in cui l'ultimo valore è identificato dal valore zero.

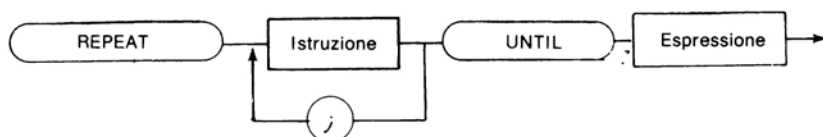
L'algoritmo che utilizza la struttura *ripetere* è il seguente:

sia max = 0
ripetere
  leggere numero
  se numero > max *allora* max = numero
finchè numero = 0
scrivere max

Il corrispondente grafo GNS è:

Il programma è il seguente:

```
PROGRAM massimo;
VAR numero, max: integer;
BEGIN
  max: = 0;
  REPEAT
    read (numero);
    IF numero > max THEN max: = numero;
  UNTIL numero = 0;
  writeln ('il massimo è =', max);
END.
```

5.7.2.1 - Sintassi dell'istruzione ripetere

È espressa dal diagramma seguente:

Dal punto di vista semantico, questa struttura ripete le istruzioni comprese fra la parola riservata REPEAT (RIPETERE) e la parola riservata UNTIL (FINCHÈ), FINCHÈ l'espressione diventa *vera*.

Quindi l'espressione deve esprimere una condizione suscettibile di variare nel corso dell'iterazione. In particolare, bisogna, ancora una volta, che il valore di questa condizione sia modificato in una delle istruzioni della struttura, altrimenti avremmo un ciclo infinito.

Si tenga presente poi che le istruzioni della struttura sono sempre eseguite *almeno una volta*.

Esempi

1)
```
REPEAT
  writeln ('io sono venuto qui')
UNTIL true;
```

Quest'istruzione scriverà il messaggio "io sono venuto qui", benché la condizione sia sempre vera.

2) Anche qui bisogna stare attenti al numero d'iterazioni realmente effettuate, tenuto conto del risultato previsto.

```
PROGRAM ripetere;
VAR nriterazione: integer;
BEGIN
nriterazione: = 1;
REPEAT
  writeln ('iterazione n', nriterazione);
  nriterazione: = nriterazione + 1
UNTIL nriterazione = 11
END.
```

Questo programma permette di compiere dieci iterazioni, ma bisogna definire la condizione di fine ciclo, con un numero d'iterazioni superiori di una unità al numero d'iterazioni desiderato.

Il programma può essere scritto anche così:

```
PROGRAM ripetere;
VAR nriterazione: integer;
BEGIN
  nriterazione: = 0;
REPEAT
  nriterazione: = nriterazione + 1;
  writeln ('iterazione n', nriterazione)
UNTIL nriterazione = 10;
END.
```

Anche qui sono state effettivamente compiute dieci iterazioni, ma, questa volta, la condizione corrisponde esattamente all'uguaglianza della variabile *nriterazione* e del valore d'arresto desiderato! Questo semplicemente perché abbiamo invertito le istruzioni dell'iterazione, ed abbiamo inizializzato la variabile a 0 prima di dare inizio all'iterazione.

5.7.2.2 - Passaggio da una struttura finquando ad una struttura ripetere

La differenza fra la struttura WHILE (FINQUANDO) e la struttura REPEAT (RIPETERE) sta nel fatto che, nel primo caso, l'iterazione *comincia* con la valutazione della condizione che permette di fermare o di continuare il ciclo, nel secondo caso la condizione viene valutata *alla fine* di ogni esecuzione dell'iterazione. Nel primo caso, perché il ciclo possa continuare, la condizione dev'essere *vera*, mentre nella struttura RIPETERE, perché l'iterazione continui, la condizione dev'essere *falsa*.

Quindi è sempre possibile passare da una struttura RIPETERE ad una struttura FINQUANDO invertendo la condizione. Perché ci sia equivalenza fra le due forme, bisogna che le istruzioni dell'iterazione vengano eseguite almeno una volta. Pertanto la struttura RIPETERE generale

```
REPEAT
   istruzione 1;
      ..........
   istruzione n
UNTIL condizione
```

è equivalente alla struttura FINQUANDO seguente:

```
istruzione 1;
   ..........
istruzione n;
WHILE NOT condizione DO
   BEGIN
      istruzione 1;
         ..........
      istruzione n
   END.
```

Osserviamo che il corpo dell'iterazione dev'essere duplicato prima della struttura WHILE (FINQUANDO), perché il test della condizione avviene prima dell'iterazione, nella struttura FINQUANDO. Bisogna dedurne che la struttura RIPETERE è da preferirsi? La risposta è: solo nel caso in cui le istruzioni del ciclo debbano venir eseguite *almeno una volta.*

Invece nella struttura

```
FINQUANDO condizione FARE
   INIZIO
      istruzioni dell'iterazione
   FINE
```

sappiamo che le istruzioni dell'iterazione non saranno eseguite se la condizione è *falsa* al momento dell'ingresso nella struttura FINQUANDO.

Allora la struttura RIPETERE equivalente è:

```
SE condizione ALLORA
RIPETERE
   istruzioni dell'iterazione
FINCHÈ condizione;
```

Qui saremo obbligati ad aggiungere un test prima della struttura RIPETERE, al fine di evitare l'esecuzione dell'iterazione nel caso in cui la condizione non sia soddisfatta al momento in cui si entra nella struttura iterativa.

5.7.2.3 - Esempi: algoritmi di M.C.D.

Il primo algoritmo che abbiamo introdotto, parlando della struttura FINQUANDO, è appunto un esempio di caso nel quale la struttura iterativa non dev'essere eseguita se la condizione è falsa all'inizio.

Avevamo le istruzioni

```
FINQUANDO a < > b FARE
                  SE a > b ALLORA a: = a − b
                  SENNÒ b: = b − a;
```

Nella struttura RIPETERE, le istruzioni equivalenti sono:

```
SE a < > b ALLORA
   RIPETERE
      se a > b ALLORA a: = a − b
      SENNÒ b: = b − a
   FINCHÈ a = b;
scrivereln ('mcd =', a);
```

Vediamo dunque che qui la struttura FINQUANDO è preferibile alla struttura RIPETERE, perché comporta un minor numero d'istruzioni.

Passiamo ora al secondo algoritmo del M.C.D., l'algoritmo di Euclide, basato sulla proprietà del M.C.D. per cui

MCD (a, b) = MCD (a modulo b, b)

Ricordiamo che il corpo del programma era il seguente:

```
r: = 1
FINQUANDO r < > 0 FARE
   INIZIO
      r: = a MOD b;
      a: = b; b: = r
   FINE;
scrivereln ('mcd =', a);
```

Con la struttura RIPETERE il programma sarebbe il seguente:

```
PROGRAMMA mcd;
VAR a, b: integer;
INIZIO
   leggere (a, b);
   RIPETERE
      r: = a MOD b; a: = b; b: = r
   FINCHÈ r = 0;
   scrivereln ('mcd =', a)
FINE.
```

Con questo secondo algoritmo la struttura RIPETERE è preferibile alla struttura FINQUANDO, perché questa volta bisogna eseguire almeno una volta il corpo dell'iterazione per determinare se $r = 0$.

AVVERTENZA

In alcuni casi, usando la struttura RIPETERE non si può evitare che sussistano rischi di errore in fase di esecuzione; questo accade ad esempio quando si ricorre ai predicati fine di linea (*eoln*) o fine di flusso (*eof*).

In questi casi si può scrivere

```
WHILE NOT eof DO
  elaborazione del flusso;
```

mentre non si può scrivere

```
REPEAT
  elaborazione del flusso
UNTIL eof
```

Qui si avrebbe inevitabilmente un errore in fase di esecuzione, allorquando si cercasse di lavorare su un record del flusso che non esiste, perché si è a fine flusso.

Pertanto, anche se, dal punto di vista sintattico, è possibile passare dalla forma FINQUANDO alla forma RIPETERE e viceversa, alcune strutture FINQUANDO non hanno un equivalente semantico diretto nella struttura RIPETERE. Nell'esempio precedente, bisognerà adottare una struttura di questo tipo:

```
RIPETERE
  SE NON eof ALLORA
    elaborazione del flusso
  SENNÒ istruzione nulla
FINCHÈ eof
```

Per questa ragione è generalmente preferibile provare ad usare la struttura FINQUANDO prima di decidersi per la struttura RIPETERE: nella maggior parte dei casi la scelta risulta evidente.

5.8 - La struttura iterativa for (per)

Le strutture precedenti sono utili solo se ignoriamo a priori il numero d'iterazioni da compiere. In caso contrario, usandole bisogna far intervenire una variabile di controllo del numero d'iterazioni.

Questa variabile dev'essere inizializzata prima della struttura iterativa ed incrementata all'interno della struttura, fintantoché non sarà uguale ad un valore limite, o finché non raggiunga il valore che indica la fine delle iterazioni.

Supponiamo ad esempio di dover programmare il problema del calcolo della somma dei primi *n* numeri. Con una struttura iterativa FINQUANDO, avremmo:

```
PROGRAM somma;
VAR n, som, i: integer;
BEGIN
  read (n);
  som: = 0; i: = 1;
  WHILE i < = n DO
    BEGIN
      som: = som + i; i: = i +1
    END;
  writeln ('somma dei primi', n, 'interi =', som);
END.
```

Qui la variabile di controllo è *i*, che viene inizializzata a 1 prima della struttura FINQUANDO, ed incrementata di 1 all'interno della struttura. L'iterazione prosegue *finquando i* non supera il valore del dato *n*.

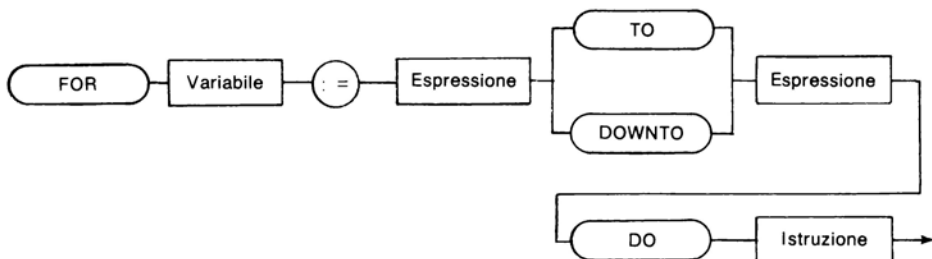
Con una struttura ciclica FOR (PER), sapendo che bisogna compiere l'iterazione *n* volte, facciamo variare *i* da 1 a *n*.

Avremo allora:

```
PROGRAM somma;
VAR n, i, som: integer;
BEGIN
  read (n); som: = 0;
  FOR i: = 1 TO n DO som: = som + i;
  writeln ('somma dei primi', n, 'interi =' som);
END.
```

Vediamo pertanto che questa struttura è più concisa, e più adatta per questo tipo di problema. L'istruzione d'incremento, che con la struttura FINQUANDO doveva essere programmata, qui è invece implicita.

5.8.1 - Sintassi dell'istruzione for (per)

La forma generale dell'istruzione PER è data dal seguente diagramma sintattico:

Questa struttura prevede anche la possibilità di incrementare o decrementare la variabile di controllo. Ciò è espresso dalle parole riservate TO e DOWNTO.

Si hanno comunque delle restrizioni di ordine *semantico*:

- La variabile di controllo dev'essere scalare, ma è escluso il tipo reale.
- L'incremento è rappresentato dal valore successivo nell'insieme del tipo associato; il decremento dal valore precedente nel medesimo insieme.

Consideriamo la forma

PER variabile: = espressione 1 A espressione 2 FARE
istruzione;

I tipi della variabile e delle espressioni 1 e 2 devono essere identici. Durante l'esecuzione, il valore dell'espressione 1 è maggiore di quello dell'espressione 2, allora l'istruzione che vien dopo la parola riservata DO (FARE) non viene eseguita.

Analogamente, nella forma

PER variabile: = espressione 1 D espressione 2 FARE
istruzione;

in cui la D sta per decrementare, l'istruzione non viene eseguita se si ha:

valore (espressione 1) $<$ valore (espressione 2)

AVVERTENZA

Attenzione! Nel Pascal standard il valore della variabile di controllo è *indeterminato all'uscita* dall'esecuzione di un ciclo iterativo PER concluso normalmente. Inoltre la variabile di controllo *non può essere modificata* all'interno del ciclo. Da ultimo, i valori iniziali e finali delle espressioni vengono *valutati, una sola volta*, al momento dell'entrata nel ciclo.

5.8.2 - Ciclo per con tipi non standard

Come abbiamo appena detto, è possibile definire cicli PER con variabili di controllo non standard. Ad esempio, definendo il tipo

TYPE nomese = (gennaio, febbraio, marzo, ..., ottobre,
novembre, dicembre);

e la variabile

```
VAR mese: nomese;
```

possiamo definire il seguente ciclo iterativo:

```
FOR mese: = gennaio TO dicembre DO istruzione;
```

5.8.3 - Strutture equivalenti

In linea generale, la struttura

```
PER variabile: = espressione 1 A espressione 2 FARE
  istruzione;
```

è equivalente a

```
SE espressione 1 < = espressione 2 ALLORA
INIZIO
  variabile: = espressione 1;
  istruzione;
  variabile: = succ (variabile);
  istruzione;
    ..........
  variabile: = espressione 2;
  istruzione
FINE;
```

ed anche alla seguente struttura FINQUANDO:

```
variabile: = espressione 1;
FINQUANDO variabile < = espressione 2 FARE
INIZIO
  istruzione;
  variabile: = succ (variabile)
FINE;
```

Analogamente, la struttura

```
PER variabile: = espressione 1 D espressione 2 FARE
  istruzione;
```

è equivalente alla struttura

```
SE espressione 1 > = espressione 2 ALLORA
INIZIO
   variabile: = espressione 1;
   istruzione;
   variabile: = pred (variabile);
   ...........
   variabile: = espressione 2;
   istruzione
FINE;
```

e, in generale, alla struttura

```
variabile: = espressione 1;
FINQUANDO variabile > = espressione 2 FARE
INIZIO
   istruzione;
   variabile: = pred (variabile)
FINE;
```

AVVERTENZA

L'equivalenza con una struttura WHILE (FINQUANDO) non è assoluta, perché nelle due strutture equivalenti appena viste la variabile di controllo ha un valore che può essere definito mediante *succ* (*espressione* 2) o mediante *pred* (*espressione* 2), mentre si è visto che questo valore è indeterminato all'uscita da una struttura PER.

5.8.4 - Annidamento di cicli per

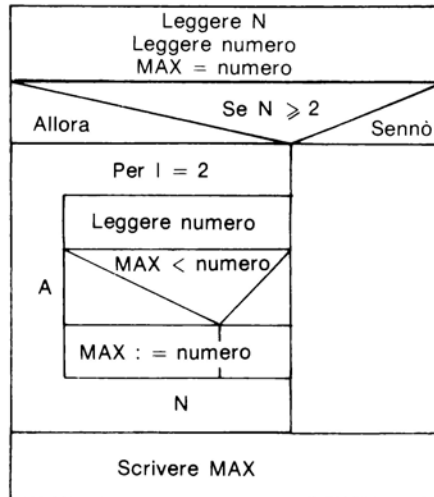
La definizione sintattica dell'istruzione PER permette di pensare a strutture cicliche annidate l'una nell'altra.

Ad esempio si possono avere strutture del tipo

```
PER v1: = e1 A e2 FARE
   INIZIO
      PER v2: = ........... FARE
         INIZIO
            PER v3: = .......... FARE ...........
         FINE;
   FINE;
```

La struttura PER più interna non deve necessariamente contenere un'istruzione composta che cominci con INIZIO e termini con FINE.

Esempio

Riprendiamo il problema della ricerca del numero massimo di una sequenza, ove sia noto il numero degli elementi da leggere. Avremo:

Non è più necessario verificare il valore di *numero* per sapere quando fermare l'iterazione, dato che il numero degli elementi da leggere è noto. Quest'algoritmo, anche se più complesso, è comunque più generale, perché permette di trovare il numero massimo di una sequenza qualunque. Va detto tuttavia che è anche più restrittivo, dal momento che impone di precisare il numero degli elementi.

Vedremo che l'impiego di booleani come *eof* ed *eoln* permette di eludere questo vincolo, così come avviene con le strutture FINQUANDO e RIPETERE.

Il programma corrispondente è:

```
PROGRAM maxper;
VAR n, numero, i, max: integer;
BEGIN
  read (n);
  read (numero);
  max: = numero;
  IF n > = 2 THEN
  BEGIN
    FOR i: = 2 TO n DO
    BEGIN
      read (numero);
      IF max < numero THEN max: = numero;
    END;
  END;
  writeln ('massimo =', max);
END.
```

Questo programma è più lungo delle versioni che utilizzano le strutture FINQUANDO o RIPETERE, il che dimostra che questa struttura non è appropriata al problema da risolvere.

5.9 - L'istruzione di salto goto (andarea)

Finora abbiamo fatto benissimo a meno di questa istruzione, e continueremo a farlo tutte le volte che sarà possibile. Comunque, in chiusura di questo capitolo dedicato agli elementi fondamentali del linguaggio, è necessario parlarne per completezza.

L'istruzione di salto richiede l'impiego di etichette, che devono essere dichiarate nella sezione corrispondente (LABEL). Ricordiamo che in Pascal un'*etichetta* è un intero non segnato formato da quattro cifre al massimo. L'etichetta precede un'istruzione, con un separatore costituito dal carattere: . Una stessa etichetta non può essere utilizzata due volte nello stesso blocco di programma.

5.9.1 - Sintassi dell'istruzione goto

La sintassi dell'istruzione GOTO (ANDAREA) è molto semplice: GOTO (ANDAREA) etichetta.

Quest'istruzione determina quindi il salto all'etichetta che vien dopo la parola riservata GOTO (ANDAREA).

Esempio

```
PROGRAM andarea;
LABEL 1;
VAR x, y: real;
BEGIN
  read (x);
  IF x < 0 THEN GOTO 1
    ELSE y: = sqrt (x);
    ..........
  1: writeln ('valore negativo impossibile =', x);
END.
```

Anche se un programma può venir scritto senza ricorrere all'istruzione GOTO, quest'ultima può essere utile quando si vuol trattare un caso di errore, e trasferire il controllo direttamente alla fine di un blocco o di un programma.

L'uso dell'istruzione ANDAREA comporta alcune restrizioni di ordine semantico: non si può saltare ad un punto qualunque e in modo qualunque! Infatti, come si è visto, il Pascal è un linguaggio strutturato. Se per mezzo di un'istruzione ANDAREA è sempre possibile uscire da una struttura, è invece impossibile con questa stessa istruzione rientrare nel bel mezzo di una struttura: questa è la limitazione più importante, e si badi che questo tipo di errore non può essere segnalato dai compilatori.

Esempi

1) È possibile uscire da una struttura di selezione:

SE condizione ALLORA ANDAREA 1 SENNÒ ...

2) Non si può invece rientrare nel mezzo di tale struttura:

SE condizione ALLORA 1: istruzione SENNÒ 2: istruzione

In questa struttura non si può utilizzare un'istruzione ANDAREA 1 o ANDAREA 2.

3) Analogamente, è possibile uscire da una struttura iterativa:

```
FINQUANDO condizione FARE
   INIZIO
   ..........
   ANDAREA 1; ...
   FINE
```

4) Invece non si può avere:

```
ANDAREA 2
..........
FINQUANDO condizione FARE
   INIZIO
   ..........
   2: istruzione; ...
   FINE;
```

E questo con qualunque struttura iterativa.

Queste regole valgono anche per i salti e le uscite da blocchi, procedure e funzioni.

OSSERVAZIONI

In molti casi si può evitare di ricorrere ad un'istruzione ANDAREA, a patto di usare variabili booleane ed istruzioni di selezione. Ma, se questo compromette in modo apprezzabile la comprensione dell'algoritmo, tanto vale usare quest'istruzione "maledetta". D'altra parte alcune applicazioni hanno mostrato che si può benissimo programmare in modo strutturato pur utilizzando istruzioni di salto.

Al lettore abituato a programmare in FORTRAN o in BASIC consigliamo pertanto di evitare all'inizio di utilizzare questa istruzione, in modo che l'apprendimento di un algoritmo strutturato possa avvenire nelle condizioni migliori.

5.9.2 - Un'estensione del Pascal U.C.S.D.: l'istruzione exit

Come si è già detto, l'istruzione di salto (GOTO) è utile quando si vuol uscire da una struttura prima della sua normale conclusione.

L'istruzione *exit* del Pascal U.C.S.D. è prevista in modo specifico per questo caso: evita di utilizzare un GOTO che rimandi ad una data etichetta, e permette altresì di uscire da una procedura o da una funzione (v. Cap. 5).

La forma più semplice di questa istruzione è costituita dall'uscita da un programma (ad esempio in caso di errore non recuperabile), ottenuta scrivendo

```
exit (program)
```

oppure

```
exit (nome del programma)
```

Quindi quest'istruzione è equivalente ad un'istruzione di salto alla fine del programma. In particolare, potremo usarla in una delle alternative di una struttura di selezione:

```
IF condizione THEN exit (program) ELSE ...
```

La forma più generale di quest'istruzione è:

```
exit (nome della procedura)
```

In questo caso l'esecuzione dell'istruzione determina l'uscita dalla procedura specificata. Così, se si hanno più livelli di procedure annidate, potremo uscire da un qualunque livello con una sola istruzione: usando invece delle istruzioni di salto, il programma ne risulterebbe molto appesantito. Su questo torneremo dopo aver parlato delle procedure.

Resta comunque il fatto che, se si vuol rispettare lo standard originario del Pascal, è opportuno non usare quest'istruzione.

5.10 - Note riassuntive sul capitolo 3

In questo capitolo abbiamo illustrato gli elementi fondamentali del linguaggio, che si possono così riassumere:

- Esistono quattro sezioni di dichiarazione, nelle quali sono dichiarate le *etichette*, le *costanti*, i *tipi* e le *variabili* utilizzati nel programma. La particolarità del Pascal è che tutti gli oggetti che si utilizzano nel corpo del programma *debbono* essere dichiarati. Si possono poi anche definire *tipi non standard.*

- In Pascal il concetto di espressione è alla base di tutte le istruzioni strutturate, e comprende tutti i tipi: aritmetico, booleano, relazionale. Anche gli operatori relazionali si applicano a tutti i tipi, compresi quelli definiti dal programmatore.
- L'istruzione di assegnazione conserva il significato che ha abitualmente in programmazione, ma, anche qui, il concetto di assegnazione si applica a tutti i tipi di espressioni.
- L'istruzione di selezione semplice è un'istruzione strutturata:

```
SE condizione ALLORA istruzione 1
              SENNÒ istruzione 2
```

  La clausola SENNÒ può essere omessa.
- Esiste un'istruzione di selezione multipla, la cui forma è:

```
CASO espressione FRA
  selezione 0: istruzione 0;
  selezione 1: istruzione 1;
     ..........
  selezione n: istruzione n;
FINE
```

Abbiamo poi esaminato in dettaglio tutte le istruzioni iterative:

```
FINQUANDO condizione FARE istruzione
RIPETERE istruzione FINCHÈ condizione
PER variabile: = valore iniziale A valore finale FARE
    istruzione
```

In tutti questi casi, l'istruzione può essere composta. La scelta fra queste strutture iterative dipende dalla natura del problema da programmare, ma è sempre possibile passare da una forma all'altra.

Infine abbiamo introdotto l'istruzione di salto, sottolineando le restrizioni che essa comporta.

5.11 - Alcuni esempi

1. Un numero perfetto è un numero caratterizzato dal fatto di essere uguale alla somma di tutti i suoi divisori, eccettuato il numero stesso. Il primo numero perfetto è 6, uguale a 1 + 2 + 3, che sono appunto i suoi divisori.
   Scrivere un programma che stampi la lista di tutti i numeri perfetti fino a 200.

2. Due numeri *m* ed *n* si dicono numeri amici se la somma dei divisori di *m* è uguale

ad n, e la somma di tutti i divisori di n è uguale ad m: ad esempio 200 e 284 sono numeri amici.
Scrivere un programma che permetta di trovare almeno un'altra coppia di numeri amici. Si badi che per trovare la prossima coppia di numeri amici bisogna arrivare fino a 2000.

3. Fra tutti gl'interi > 1, solo quattro possono venir rappresentati con la somma dei cubi delle cifre che li compongono. Ad esempio, $153 = 1^3 + 5^3 + 3^3$.
Scrivere un programma che determini gli altri tre. Si badi che tutti e quattro i numeri sono compresi fra 150 e 410.

4. Scrivere un programma che calcoli il fattoriale di n, cioè il prodotto dei primi n numeri interi.

5. Scrivere un programma che calcoli i primi 20 numeri di Fibonacci, rappresentati dalla formula di ricorrenza

$$f_{n+2} = f_{n+1} + f_n$$

in cui $f_0 = 0$, $f_1 = 1$, $n \geq 0$.
I primi numeri della sequenza sono 0, 1, 1, 2, 3, 5, 8, 13 ...

Soluzioni

1. Programma dei numeri perfetti:

```
PROGRAM perfetto;
VAR n, i, j: integer;
     somma: integer;
BEGIN
   read (n);
   FOR i: = 2 TO n DO
   BEGIN
   somma: = 1;
   FOR j: = 2 TO i DIV 2 DO
      BEGIN
         IF i/j = i DIV j THEN somma: = somma +j
      END;
   IF somma = i THEN writeln (i, 'e' 'perfetto')
   END;
END.
```

Esecuzione

```
30 (R)
 6  è perfetto
28  è perfetto
```

2. Programma dei numeri amici:

```
PROGRAM perfettoamico;
VAR n, i, j: integer;
     somma: integer;
     sommab: integer;
BEGIN
  read (n);
  FOR j: = 2 TO i DIV 2 DO
  BEGIN
  somma: = 1;
  FOR j: 2 TO i DIV 2 DO
    BEGIN
      IF i/j = i DIV j THEN somma: = somma + j
    END;
  IF somma = i THEN writeln (i, 'e' 'perfetto')
  ELSE
    BEGIN
      IF somma < i THEN
      BEGIN
        sommab: = 1;
        FOR j: = 2 TO somma DIV 2 DO
        BEGIN
          IF somma/j = somma DIV j THEN sommab: = sommab + i
        END;
        IF sommab = i THEN writeln (i, '', somma, 'sono numeri amici')
      END;
    END;
  END;
END.
```

Esecuzione

```
300 (R)
  6  è perfetto
 28  è perfetto
284  220 sono numeri amici
```

3. Programma dei numeri cubi.

```
PROGRAM numerocubo;
VAR i, j, k: integer;
     n, ncubo: integer;
BEGIN
  FOR i: = 1 TO 9 DO
    BEGIN
      FOR j: = 0 TO 9 DO
        BEGIN
          FOR k: = 0 TO 9 DO
            BEGIN
              n: = i * 100 + j * 10 + k;
              ncubo: = i * i * i + j * j * j + k * k * k;
              IF n = ncubo THEN writeln (n);
            END;
        END;
    END;
END.
```

Esecuzione

```
153
370
371
407
```

4. Si tratta di un programma simile a quello che calcola la somma dei primi *n* numeri interi:

```
PROGRAM fattoriale;
VAR fatt, n, i: integer;
BEGIN
  read (n); fatt: = 1;
  FOR i: = 2 TO n DO fatt: = fatt * i;
  writeln ('fattoriale', n, '=', fatt)
END.
```

5. Il programma che calcola i numeri di Fibonacci è il seguente:

```
PROGRAM Fibonacci;
VAR i, f0 f1, f2: integer;
BEGIN
   f0: = 0; f1: = 1;
   writeln ('numeri di Fibonacci');
   writeln (f1);
   FOR i: = 1 TO 20 DO
      BEGIN
      f2: = f1 + f0; f0: = f1; f1: = f2;
      writeln (f2);
      END;
END.
```

Esecuzione

```
numeri di Fibonacci
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
```

ESERCIZI

1. Scrivere una dichiarazione di costante per definire i valori 100, 2.73, 5%, 10^{-9}, $2\sqrt{3}$ e le parole *buongiorno*, *signora*, *signore*, *l'etere*, *troppo pieno*,*totale = prezzo + iva 33%.*

2. Definire le dichiarazioni necessarie per scrivere un programma che calcoli la superficie ed il volume di una sfera.

3. Definire una dichiarazione di tipo riferita ai principali fiumi italiani, alle sigle delle targhe automobilistiche ed all'insieme delle vocali.

4. Si abbiano le seguenti dichiarazioni:

```
CONST x = 2; c = 'cento';
TYPE eta = 0... 120;
VAR y: integer; etap: eta;
      l1: boolean;
      e: real;
      car: char;
```

Sono valide le seguenti istruzioni di assegnazione?

```
y: = x; car: = c; e: = eta; y: = etap; l1: = car;
etap: = y; cento: = c; x: = e; etap: = c;
```

5. Date le seguenti dichiarazioni:

```
CONST bianco = ' '; uno = 1;
        testo = 'taratata';

TYPE
   complesso = (reale, immaginario);
   capitale = (Bonn, Bruxelles, Copenaghen, Londra, Amster-
               dam, Parigi, Roma)

VAR x, y: real;
      i, j, k: integer;
      z: complesso; citta: capitale;
      car: char; predicato: boolean; c: string;
```

Dire se le seguenti istruzioni di assegnazione sono corrette dal punto di vista sintattico e dal punto di vista semantico:

```
z: = x + i * y;
car: = uno + bianco;
j: = ord (Parigi) + ord (Roma);
predicato: = x + y = z;
```

```
citta: = Bonn + Londra;
predicato: = z < immaginario AND reale;
car: = c + bianco;
c: = testo;
x: = y + x * j DIV i;
k: = ord (citta) + uno;
predicato = (citta < Parigi) OR (c > Londra);
```

Attenzione: ciascuna istruzione dev'essere valutata individualmente.

6. Scrivere in Pascal le dichiarazioni e le istruzioni di assegnazione corrispondenti alle espressioni seguenti:

   f = m·gamma

   $$v = \frac{4 \cdot pi \cdot r^3}{3}$$

   p = not (a and b) or (c and not d)
   condizione = a > b or d = c
   identità = nome
   esiste = $b^2 - 4ac \geq 0$
   parola = "un'impresa rischiosa non annullerà mai il caso"
   carattere = 'd'
   se x = y, allora l'uguaglianza è vera, sennò l'uguaglianza è falsa (Questa frase si può esprimere sotto forma di assegnazione).

7. Esprimere le frasi seguenti sotto forma di espressioni booleane:
   a) Per essere italiani, bisogna essere nati in Italia da genitori italiani, o esser stati naturalizzati.
   b) La presenza del sintomo di Koplick porta alla diagnosi di morbillo.
   c) Ho fretta, e ci sono ingorghi di traffico (o la mia macchina ha un guasto, o la benzina è troppo cara). Tutto ciò porta alla proposizione: Prendo la metropolitana.
   d) Non c'è fumo senza arrosto.

8. Si abbia la seguente istruzione di selezione:

```
SE x < y ALLORA SE z > 5 ALLORA a: = 4
SENNÒ SE z < 8 ALLORA SE y > 7 ALLORA a: = 1
SENNÒ SE z > 10 AND z < 20 ALLORA a: = 15
SENNÒ a: = b SENNÒ a: = x
SENNÒ a: = y;
```

L'istruzione è sintatticamente corretta?
Scrivere una forma equivalente, che sia più leggibile, ed una forma più semplice.

9. Date tre variabili *x*, *y*, *z*, scrivere un programma che permetta di attribuire ad *x* il valore più alto e ad *y* quello più basso.

10. Scrivere in una forma più semplice le istruzioni di selezione seguenti:

```
a) SE x > a ALLORA SE y < b ALLORA SE z <> 0
   ALLORA u: = y - x/z;
b) se x < a ALLORA INIZIO SE y > b
                               ALLORA u: = x + y
                          FINE
                  SENNÒ SE z < > 0
                         ALLORA u: = (x - y)/z;
```

11. Scrivere un programma capace di leggere due dati *n* ed *x*, e di scrivere, ricorrendo all'istruzione CASO ... FRA, il risultato fornito dalle funzioni matematiche standard in Pascal. Si tenga conto del fatto che alcune di queste funzioni non sono definite su tutti gli intervalli. Il dato *n* permette di selezionare la funzione, *x* è il parametro della funzione.
12. La "prova perduta" di Fermat.
    Il matematico Fermat ha ritenuto di aver dimostrato un teorema del quale non si è mai trovata la prova, ma che non si è mai potuto confutare.
    Il problema è provare che non è possibile trovare dei valori interi *a*, *b*, *c* tali per cui $a^n + b^n = c^n$, per $n > 2$.

    a) Al fine di verificare il teorema di Fermat su un sottoinsieme di numeri interi, si chiede di scrivere un programma che permetta di far variare *a*, *b*, *c* fra 1 e 30 per tutti i valori di *n* compresi fra 3 e 5.
    b) Determinare inoltre tutte le terne, *a*, *b*, *c* tali per cui $1 \leq a < b < c$, per le quali si ha:
    $$a^n + b^n - c^n \leq 15$$

13. Scrivere un programma che permetta di calcolare gl'interessi maturati mensilmente relativi ad un capitale *c* investito al tasso dell'*i*% annuo.

14. Scrivere un programma che permetta di trovare il numero di termini della serie armonica necessario per superare un valore arbitrario dato:

$$1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n > \text{valore}$$

15. Riscrivere il programma precedente per calcolare la somma dei primi *n* termini della serie armonica dell'Esercizio 14.

16. Scrivere dei programmi che calcolino i valori delle serie seguenti, per *x* = 0, 0.5, 1, 2, 2.7, 3.14.

$$\sum_{n=0} \frac{x^n}{n!}$$

$$\sum_{n=0} (-1)^n x^n$$

$$\sum_{n=0} (-1)^{n-1} \frac{x^n}{n}$$

$$\sum_{n=0} (-1)^n \frac{x^{2n} + 1}{(2n + 1)!}$$

17. Scrivere un programma in Pascal che permetta di calcolare *x* con la precisione di 10^{-5}, ricorrendo alle due sequenze *An* e *Bn* definite dalle relazioni

$$A_{n+1} = \frac{An + Bn}{2}$$

$$B_{n+1} = \frac{2AnBn}{An + Bn}$$

dove $A_0 = X$ e $B_0 = 1$.

Assumiamo dimostrato che $\lim An = \lim Bn = \sqrt{X}$ quando $n \to \infty$

18. Scrivere un programma che calcoli il minimo comune multiplo (m.c.m.) di due numeri interi, *n* ed *m*. Verificare con un programma la relazione

$$\text{mcm } (n, m) \cdot \text{MCD } (n, m) = n \cdot m$$

19. Sapendo che la somma

$$1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 - 1/15 + \ldots$$

tende a $\pi/4$, scrivere un programma che permetta di calcolare π con 4 decimali.

20. Calcolo di *e*.
e è così chiamato in onore di Eulero, che ha dimostrato che *e* è esprimibile sotto forma di una frazione del tipo

$$n_1 + \cfrac{1}{n_2 + \cfrac{1}{n_3 + \cfrac{1}{n_4 + \ldots\ldots}}}$$

che prende il nome di frazione continua $(n_1, n_2, n_3, n_4 \ldots)$.
Scrivere un programma che permetta di calcolare *e*, sapendo che è rappresentato dalla frazione continua

$$e = (2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, \ldots)$$

CAPITOLO 4

I DATI STRUTTURATI IN PASCAL: VETTORI ED INSIEMI

"Ci si convince di più, solitamente, con le ragioni trovate da noi stessi che con quelle nate nella mente degli altri".

PASCAL – Pensées

1 - I DATI STRUTTURATI

Come si è visto, il linguaggio Pascal è strutturato in quanto le sue istruzioni permettono di programmare algoritmi espressi in forma strutturata.

D'altra parte, trattando le sezioni di dichiarazione dei dati e delle variabili, abbiamo parlato soltanto dei dati scalari, che rappresentano un valore di tipo semplice, standard o non standard.

In questo capitolo introdurremo nuovi tipi di dati a cui vengono attribuite strutture più complesse, in particolar modo il tipo ARRAY (VETTORE) ed il tipo SET (INSIEME). Quanto al tipo RECORD, se ne parlerà nel Capitolo 6.

Ciascun elemento di queste strutture di dati può, all'occorrenza, essere considerato come un dato semplice, appartenente ad un tipo fra quelli già definiti nei capitoli precedenti. Quindi questi dati si potranno utilizzare nelle istruzioni eseguibili di cui si è già parlato. Vedremo tuttavia che, nel caso dei tipi insieme e record, si devono introdurre nuovi operatori e nuove istruzioni.

L'utilità di queste strutture di dati sta nel fatto che esse permettono di definire non solo degli oggetti matematici noti, quali vettori, matrici, insiemi, ma anche oggetti indispensabili in alcuni tipi di elaborazioni: questo è, ad esempio, il caso dei records, che intervengono nella strutturazione dei flussi, di cui, come si è già detto, parleremo in un capitolo successivo.

1.1 - Il concetto di vettore

Gli elementi fondamentali del linguaggio considerati finora erano tutti delle variabili semplici, che, in un momento dato, contenevano un solo valore.

Ora, spesso è necessario lavorare su insiemi di dati strutturati: una sequenza di numeri a_i, una sequenza di parole contenute in un dizionario p_i, un vettore, etc.

Anche il trattamento di particolari problemi matematici richiede molto spesso l'impiego di strutture matriciali, cioé con righe e colonne, a cui corrispondono elementi del tipo a_{ij}.

In matematica, le variabili di tipo a_i, a_{ij} prendono il nome di elementi indicizzati o generici.

1.1.1 - Il concetto di variabile indicizzata

In programmazione non si possono definire gl'indici mediante un artificio tipografico. È invece possibile definire degl'indici associandoli a nomi di variabili. Parleremo allora di *variabili indicizzate*. Ad esempio, a_i è rappresentata da *a* [*i*], dove *a* ed *i* sono identificatori, ed *i* rappresenta un numero intero. *a* [*i*] è una variabile indicizzata: rappresenta il generico elemento di un vettore formato dagli elementi *a*[*1*], *a*[*2*], *a*[*3*], ... *a*[*k*] ... *a*[*n*], dove *n* è l'indice associato all'ultimo elemento del vettore in questione. Di fatto, in programmazione non esiste una lista infinita di elementi, per cui bisogna *dimensionare* il numero massimo possibile degli elementi di un vettore.

Analogamente, per rappresentare un elemento con doppio indice, come b_{ij}, definiremo una variabile indicizzata *b*[*i,j*], dove *b* è un identificatore di variabile, ed *i* e *j* sono variabili numeriche *intere*. Allora *b*[*i,j*] rappresenta il generico elemento di un vettore costituito dagli elementi *b*[*1,1*], *b*[*1,2*], *b*[*1,3*] ... *b*[*1,n*], *b*[*2,1*] ... *b*[*2,n*] ... *b*[*m,1*] ... *b*[*m,n*]. Qui *n* è l'indice che rappresenta l'ultima colonna del vettore, ed *m* è l'indice che rappresenta l'ultima riga. Gli elementi di questo vettore si possono rappresentare come segue:

```
b[1,1]   b[1,2] ... b[1,n]
b[2,1]   b[2,2] ... b[2,n]

b[m,1]   b[m,2] ... b[m,n]
```

Se $n = m$, si ha un vettore quadrato.

I vettori con un indice e con due indici permettono di rappresentare rispettivamente i vettori e le matrici in senso matematico, ma tuttavia non bisogna confonderli con questi. Un vettore in Pascal può essere costituito da elementi che non sono elementi di una matrice: è infatti possibile definire vettori di stringhe di caratteri, o di qualunque altro tipo.

OSSERVAZIONI

Il vettore con un solo indice è detto anche vettore ad una dimensione.

1.2 - Dichiarazione dei vettori in Pascal

Per poter utilizzare in un programma delle variabili indicizzate, bisogna prima dichiarare in modo esplicito che si tratta di un vettore di valori appartenenti ad un medesimo tipo. In Pascal un vettore può essere considerato come un *tipo strutturato*: come tale, può venir dichiarato, o definito, in una dichiarazione di tipo, oppure direttamente in una dichiarazione di variabile, all'interno della parte che definisce il tipo della variabile stessa.

Un vettore è una lista di valori individuati da un indice. Pertanto è necessario precisare sia il tipo dell'indice, sia il tipo degli elementi del vettore. L'indice, detto anche *selettore*, non è necessariamente quello che s'intende abitualmente con il termine "indice", perché può essere definito come appartenente al tipo scalare, o sottocampo, ma non ai tipi intero e reale.

Pertanto in una dichiarazione di tipo scriveremo:

```
TYPE lista = ARRAY [tipoindice] OF tipolista;
```

Nella versione italiana, generalmente la parola ARRAY si traduce in VETTORE, traduzione che peraltro lascia alquanto a desiderare. Avremo allora:

```
TIPO lista = VETTORE [tipoindice] DI tipolista;
```

L'indice si rappresenta fra parentesi quadre. Nella dichiarazione è indicato il tipo dell'indice, non la sua dimensione, come avviene in altri linguaggi di programmazione. Nella maggior parte dei casi questo tipo è costituito da un tipo sottocampo, in particolare sottocampo di valori interi.

Esempi

1) TYPE lista = ARRAY [0 ... 10] OF real;

o anche

```
TIPO vettormat = VETTORE [0 ... max] DI reale;
```

In questi due casi, l'indice è del tipo sottocampo, definito per mezzo di un limite inferiore ed un limite superiore espressi con interi.

2) Ma si può anche definire:

```
TYPE
   coordinata = (a, b, c);
   vettormat = ARRAY [coordinata] OF real;
```

In questo caso coordinata è un tipo non standard che definisce i nomi delle coordinate in questione.

3) I vettori possono anche essere definiti direttamente, nella sezione di dichiarazione delle variabili.

```
VAR
   sequenza: ARRAY [0 ... n] OF integer;
   stringa: ARRAY [0 ... k] OF char;
```

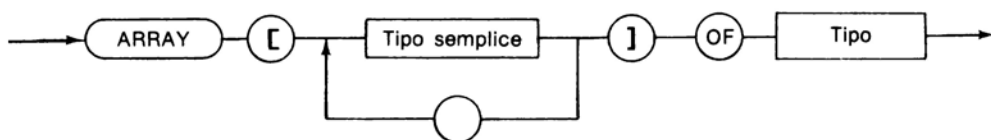
4) Ma si può avere anche

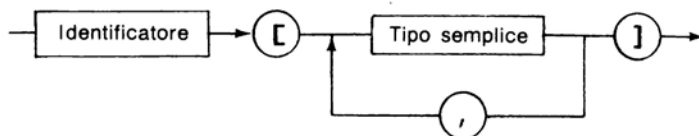
```
TYPE
   sintomo = (febbre, nausea, astenia, delirio);
VAR
   paziente: ARRAY [sintomo] OF boolean;
```

Il vettore *paziente* è una sequenza di valori che possono assumere i valori *vero* o *falso* a seconda che siano presenti o no i segni definiti nel tipo *sintomo*. Quindi, nel corso del programma, potremo avere istruzioni del tipo

```
paziente [febbre]: = vero (true)
paziente [delirio]: = falso (false)
```

1.2.1 - La sintassi

La sintassi di una *dichiarazione di vettore* è parte del diagramma sintattico di un tipo (v. Appendice 2), e si può rappresentare in questo modo:

Vedremo più avanti che esiste un altro modo per definire un vettore in forma compatta.

La sintassi di una *variabile indicizzata* corrispondente ad un elemento di un vettore è espressa dal diagramma seguente:

È chiaro che le espressioni usate devono essere dello stesso tipo di quelle che sono state definite nella dichiarazione di vettore.

OSSERVAZIONI

I vettori sono strutture di dati dette *ad accesso casuale*, perché in essi il tempo di accesso ad un elemento è indipendente dal posto occupato dall'elemento nella struttura.

1.3 - I vettori a più dimensioni

I diagrammi sintattici riportati nel paragrafo precedente mostrano che gl'indici possono essere più d'uno, e di conseguenza rappresentano dei vettori nei quali ciascun elemento è identificato da diversi valori degl'indici. In questo caso parliamo di vettori a più dimensioni.

Inoltre, nella misura in cui gli elementi di un vettore possono essere di qualsiasi tipo, è possibile definire un vettore i cui elementi siano anch'essi di tipo strutturato in particolari vettori: questo è un altro modo di rappresentare un vettore a più dimensioni.

Esempi

```
VAR
  matrice: ARRAY [0 ... n, 0 ... m] OF real;
```

si può definire anche così:

```
matrice: ARRAY [0 ... n] OF ARRAY [0 ... m] OF real;
```

o ancora così:

```
TYPE
  indiceriga = 0 ... n;
  indicecolonna = 0 ... m;
  colonna = ARRAY [indicecolonna] OF real;
  mat = ARRAY [indiceriga] OF colonna;
VAR
  matrice: mat;
```

Le dichiarazioni fatte all'inizio sono più concise, e sono da preferire quando il programma comporta una sola dichiarazione di questo tipo. La dichiarazione vista per ultima è invece preferibile quando si devono definire più variabili dello stesso tipo.

Pertanto, per dichiarare più matrici della stessa dimensione, basterà scrivere:

```
VAR
   mat1, mat2, mat3, mat4: mat;
```

Tuttavia, nel modo di scrivere le corrispondenti variabili indicizzate si nota una leggera differenza:

- quando il vettore è dichiarato con più indici, scrivendo *matrice* [*i, j*] intenderemo rappresentare il generico elemento del vettore;
- quando un vettore a più dimensioni è definito in forma ricorsiva, scriveremo: *matrice* [*i*] [*j*].

1.3.1 - Esempio di ricerca degli elementi minimo e massimo di una lista

Introduciamo *n* dati interi in un vettore denominato *num*. Il numero degli elementi è noto, per cui ricorreremo a delle strutture iterative PER; se invece il numero degli elementi non fosse noto a priori, useremmo la struttura FINQUANDO.

L'algoritmo è estremamente semplice, quindi possiamo presentare direttamente il programma, che è questo:

```
PROGRAM massimominimo;
CONST n = 10;
VAR num: ARRAY [1 ... n] OF integer;
      i, max, min: integer;
BEGIN
   writeln ('introdurre', n, 'interi');
   FOR i: = 1 TO n DO read (num[i]);
   max: = num [i]; min: = max;
   FOR i: = 1 TO n DO
   BEGIN
      IF max < num [i] THEN max: = num [i];
      IF min > num [i] THEN min: = num [i];
   END;
   writeln ('il massimo è'', max);
   writeln ('il massimo è'', min);
END.
```

1.4 - Le liste lineari: applicazione degli algoritmi di ordinamento

Le statistiche condotte su un gran numero di installazioni di calcolatori mostrano che in media il 25% del tempo di elaborazione è dedicato all'ordinamento degli elementi di una lista lineare; questa percentuale può arrivare al 50% in alcuni casi di calcolatori ad uso gestionale. Le motivazioni di questo fatto sono tre:

- moltissime applicazioni (e soprattutto il trattamento dei flussi) utilizzando programmi di ordinamento;
- molti utilizzatori ricorrono a programmi di ordinamento, anche quando non è necessario;
- molti programmi di ordinamento sono fatti in modo non efficace.

Di conseguenza, è importante conoscere i diversi metodi ed algoritmi di ordinamento.

1.4.1 - Presentazione del problema

Si abbiano *n* elementi *r1*, *r2*, ... *rn*, che chiameremo *records* (v. oltre): questi elementi sono costituiti da entità, articoli, etc. contenuti in un flusso. A ciascun *ri* è associata una chiave *ci*. Ci proponiamo di classificare questi records in modo tale che le chiavi *c1* ... *cn* associate ad essi siano ordinate.

Per far questo bisogna aver prima definito una relazione d'ordine, in base alla quale due chiavi *c1* e *c2* obbediscano sempre ad una di queste relazioni: $c1 < c2$; $c1 = c2$; $c1 > c2$.

Scopo di un algoritmo di ordinamento è determinare le permutazioni *p(1)*, *p(2)* ... *p(n)* dei records *r1* ... *rn* tali per cui $c(1) \leq c(2) \ldots \leq c(n)$.

Diciamo che l'ordinamento è *stabile* quando records corrispondenti a chiavi identiche conservano il loro ordine primitivo.

Possiamo distinguere due categorie fondamentali di ordinamento:

- l'*ordinamento interno*, in cui tutti i records sono nella memoria centrale, per cui l'algoritmo non richiede nessuna operazione d'ingresso/uscita;
- l'*ordinamento esterno*, in cui soltanto una parte dei records può essere messa nella memoria centrale. L'ordinamento avverrà necessariamente per fasi successive, ed opererà su dei sottoinsiemi di records posti nella memoria centrale. Quindi nell'ordinamento esterno intervengono dei metodi di ordinamento interno, associati ad operazioni d'ingresso/uscita.

1.4.2 - L'ordinamento interno

Esistono diverse categorie di ordinamento interno:

- *ordinamento per inserimento*: ciascun elemento è inserito nella posizione esatta, nell'ambito della lista di elementi sottoposta ad ordinamento;
- *ordinamento per scambio*: due elementi classificati in modo inesatto sono rimessi in ordine scambiandone le posizioni nella sequenza degli elementi sulla quale si svolge l'ordinamento;
- *ordinamento per selezione*: l'ordinamento avviene cercando ogni volta l'elemento più grande (o quello più piccolo);
- *ordinamento per enumerazione*: ogni elemento è confrontato con i rimanenti. Il numero delle chiavi inferiori a quella dell'elemento sottoposto al confronto determina la posizione dell'elemento in questione nella sequenza.

L'ordinamento interno può essere svolto operando direttamente sui records, che verranno messi in ordine via via che procede l'ordinamento, oppure, al contrario, operando su tabelle di puntatori associati ai records: in quest'ultimo caso i records conservano la loro posizione primitiva, e viene ordinata soltanto la tabella. Quest'operazione è detta ordinamento sulla tabella degl'indirizzi.

Una terza soluzione si ha quando si dispone di una struttura a lista organizzata in modo che la lista venga percorsa secondo l'ordine crescente (o decrescente) di una determinata chiave.

1.4.3 - Il concetto di tempo di elaborazione in relazione ad un algoritmo di ordinamento

Esistono vari tipi di algoritmi di ordinamento, per cui è utile confrontarne l'efficacia dal punto di vista del tempo di ordinamento.

Posto che si disponga di n records, dev'essere possibile qualificare un algoritmo per mezzo di una funzione del tipo $f(n)$, che dia l'ordine di grandezza del tempo di calcolo necessario per la realizzazione dell'ordinamento.

Diremo che un ordinamento richiede un tempo dell'ordine di n^2 (indicato con $o(n^2)$), se il tempo di calcolo è, in media, proporzionale al quadrato del numero di elementi da classificare.

Gli algoritmi meno efficaci sono dell'ordine di n^2; i più efficaci dell'ordine di $n \cdot log(n)$.

Vedremo degli esempi di algoritmi per ogni categoria di ordinamento definita precedentemente, limitandoci però ad ordinamenti operati su numeri, dal momento che tutte le chiavi possono essere considerate come codici binari.

1.4.4 - Ordinamento per selezione

Supponiamo di dover risolvere il problema della classificazione in ordine crescente di una sequenza di numeri. Esistono diversi algoritmi di ordinamento: uno dei più semplici, ma anche dei meno efficaci come tempo-macchina, cerca il valore minimo e lo colloca come primo elemento della lista, quindi cerca il valore minimo successivo e lo colloca al secondo posto, e così di seguito.

Supporremo che il numero degli elementi della lista sia noto.

L'algoritmo si basa sulla ricerca del valore minimo di una lista già presente. Il valore minimo, una volta trovato, dev'essere tolto dalla lista, memorizzando il relativo numero d'ordine, in modo da scambiarne il posto con quello del primo elemento della lista. Avremo allora il programma:

```
PROGRAM ordinamin;
VAR c: ARRAY [1 ... 100] OF integer;
      n, i, j, k, e, min: integer;
BEGIN
  writeln ('introdurre il numero dei dati');
  read (n);
  writeln ('introdurre', n, 'dati');
  FOR i: = 1 TO n DO read (c[i]);
  FOR i: = 1 TO n DO
  BEGIN
    min: = c[i];
    FOR j: = i TO n DO
    BEGIN
    IF c[j] < min THEN
      BEGIN
        k: = j; min: = c[j];
      END;
    END;
    e: = c[i]; c[i]: = c[k]; c[k]: = e;
    write (c[i], ' ');
  END;
END.
```

Quest'algoritmo è scarsamente efficace: infatti resta di ordine n^2 qualunque sia la sequenza di numeri.

1.4.5 - Ordinamento per enumerazione

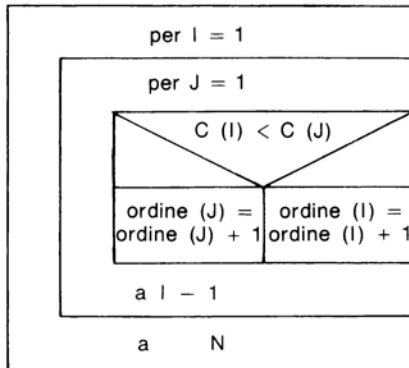
Presentiamo qui un esempio di ordinamento tramite conteggio.

Il metodo è il seguente: tutte le chiavi *cj* (con *j* che varia da 1 ad *n*) vengono confrontate con ciascuna chiave *ci*, al fine di ottenere il numero $n(j)$ di chiavi, tali per cui $ci \leqslant cj$, con *i* che varia da 1 ad *n*.

Esauriti tutti i confronti, la *k-esima* chiave sarà tale per cui per tutti gli $j < k$, $n(j)$ sarà minore di $n(k)$.

L'algoritmo è scarsamente efficace in quanto confronta due volte le stesse chiavi (*ci* e *cj*).

Di conseguenza il ciclo interno si arresta ad *i - 1*.

L'algoritmo è il seguente:

Il programma corrispondente è:

```
PROGRAM ordinaenumerazione;
VAR c: ARRAY [1 ... 100] OF integer;
     ordine: ARRAY [1 ... 100] OF integer;
     n, j, i: integer;
BEGIN
  writeln ('introdurre il numero dei dati');
  read (n);
  writeln ('introdurre', n, 'dati');
  FOR i: = 1 TO n DO
    BEGIN ordine [i]: = 1; read (c[i]); END;
  FOR i: = 1 TO n DO
  BEGIN
    FOR j: = 1 TO i - 1 DO
    BEGIN
      IF c[j] < c[i] THEN
         ordine[i]: = ordine [i] + 1
      ELSE
         ordine[j]: = ordine[j] + 1
    END;
  END;
    FOR i: = 1 TO n DO write (ordine[i], ' ');
    writeln;
END.
```

Anche qui il tempo di elaborazione è proporzionale ad n^2.

Caso particolare: se cj n, e se $m \leq cj \leq M$, essendo $1 \leq j \leq n$, si può calcolare con una sola scansione:

ordine(m) .... ordine(M)

Se la differenza $M - m$ è piccola, si potrà avere un ordinamento abbastanza veloce.

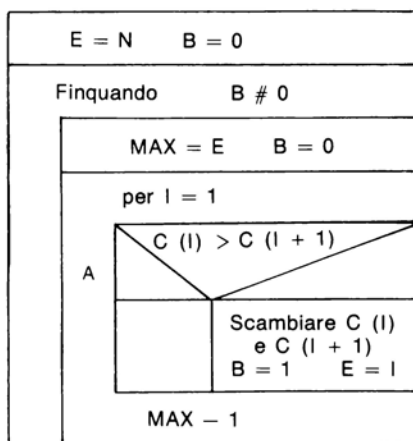
1.4.6 - Ordinamento per scambio o trasposizione

Questi algoritmi scambiano gli elementi a due a due, disponendoli in un ordine diverso.

Se consideriamo due chiavi, $c1$ e $c2$, tali per cui $c1 > c2$, volendo un ordinamento in ordine crescente scambieremo $c1$ e $c2$. Se invece le due chiavi sono già nell'ordine giusto, le lasceremo al loro posto. Estendendo il procedimento a tutte le chiavi, si otterrà una nuova classificazione, più vicina a quella definitiva.

È chiaro comunque che per realizzare un ordinamento completo bisogna eseguire diversi passaggi su tutto l'insieme dei dati.

Fra gli algoritmi di questo tipo, il più semplice, e anche il più conosciuto, è quello detto *ordinamento a bolle* perché gli elementi collocati nel posto sbagliato risalgono nella lista come le bolle alla superficie di un liquido.

L'algoritmo corrispondente è quindi:

La variabile *e* serve a memorizzare il numero degli elementi non ancora classificati. La variabile *b* indica la realizzazione di almeno uno scambio durante una fase.

Questo è il programma corrispondente:

```
PROGRAM ordinabolla;
VAR c: ARRAY[1 ... 100] OF integer;
      n, e, max, b, i: integer;
BEGIN
   writeln ('introdurre il numero dei dati');
   read (n);
   writeln ('introdurre', n, 'dati');
   FOR i: = 1 TO n DO read (c[i]);
   e: = n; b: = 1;
   WHILE b < > 0 DO
      BEGIN
         max: = e; b: = 0;
         FOR i: = 1 TO max - 1 DO
         BEGIN
            IF c[i] > c[i + 1] THEN
            BEGIN
            b: = c[i]; c[i]: = c[i + 1];
            c[i + 1]: = b; e: = i; b: = 1;
            END;
         END;
      FOR i: = 1 TO n DO write (c[i], '   ');
      writeln;
      END;
   FOR i: = 1 TO n DO writeln (c[i]);
END.
```

Esempio di esecuzione

```
introdurre il numero dei dati
5 Ⓡ
introdurre 5 dati
4 Ⓡ
7 Ⓡ
2 Ⓡ
0 Ⓡ
8 Ⓡ
4 2 0 7 8
2 0 4 7 8
0 2 4 7 8
0 2 4 7 8
0 2 4 7 8
0
```

2
4
7
8

1.4.7 - Ordinamento per inserimento

Questo metodo è detto a volte *ordinamento del giocatore di carte* perchè riproduce il metodo col quale si mettono in fila le carte: alcune vengono disposte in ordine, e poi si inseriscono le altre in modo opportuno.

Supponiamo che $r1 \ldots r_{j-1}$ siano classificati in modo che

$$c1 \leqslant c2 \ldots \leqslant c_{j-1}$$

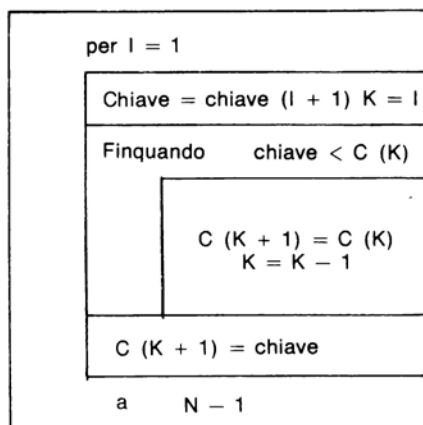
Ora, la nuova chiave *cj* viene confrontata con le chiavi già ordinate. Quando

$$ck < cj < c_{k+1}$$

il record *rj* viene inserito nella posizione corrispondente.

In pratica, questo vuol dire che si possono avere degli spostamenti di records contemporaneamente ai confronti.

Anche quest'algoritmo è scarsamente efficace, e, per di più, proporzionale ad n^2.

Per non essere costretti a cercare il valore minimo prima di far partire l'algoritmo, supporremo che il primo elemento della lista sia inizializzato con il minimo valore negativo, cioè *-maxint*. L'algoritmo allora sarà il seguente:

E questo è il programma:

```
PROGRAM ordinainserire;
VAR c: ARRAY [0 ... 100] OF integer;
     n, i k, chiave: integer;

BEGIN
  writeln ('introdurre il numero delle chiavi');
  read (n);
  writeln ('introdurre', n, 'valori');
  FOR i: = 1 TO n DO read (c[i]);
  c[0]: = -maxint: (*criterio di arresto con valore minimo*)
  FOR i: = 1 TO n - 1 DO
  BEGIN
    chiave: = c[i + 1]; k: = i;
    WHILE chiave < c[k] DO
    BEGIN
      c[k + 1]: = c[k];
      k: = k - 1
    END;
    c[k + 1]: = chiave;
  END;
  FOR i: = 1 TO n DO writeln (c[i]);
END.
```

1.4.8 - Alcuni esempi

1. Scrivere un programma che permetta di comporre la tavola pitagorica.

2. I coefficienti del binomio possono venir calcolati in maniera iterativa, ed essere rappresentati sotto forma di triangolo di Pascal, le prime righe del quale sono:

```
1
1 1
1 2 1
1 3 3 1
1 3 6 4 1
```

Calcolare le altre righe.

1. Il problema della tavola pitagorica è immediato:

```
PROGRAM tavolapit;
VAR i, j: integer;
   tav: ARRAY[0 ... 10,0 ... 10] OF integer;
BEGIN
  FOR i: = 1 TO 10 DO
  BEGIN
    FOR j: = 1 TO 10 DO
    BEGIN
      tav[i,j]: = i * j
      write (tav[i,j], '   ')
    END;
    writeln;
  END;
END.
```

In uscita si ha:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

2. Il triangolo di Pascal.

Sappiamo che il triangolo di Pascal si ottiene ponendo gli elementi della prima colonna uguali ad 1: $a_{11} = 1 \ldots a_{ij} = 1$. Gli altri valori risultano dalla relazione:

$$a_{ij} = a_{i-1j} + a_{i-1j-1}$$

ove i,j > 1 e j $\leq$ i.

In altre parole, un elemento è uguale alla somma dell'elemento che gli sta sopra e

dell'elemento posto a sinistra di quest'ultimo. Supporremo che gli elementi di una riga non definiti siano nulli. Il programma corrispondente è quindi:

```
PROGRAM Pascaltriangolo;
CONST n = 10;
VAR triangolo: ARRAY [1 ... n, 1 ... n] OF integer;
      i, j: integer;
BEGIN
   triangolo [1,1]: = 1;
   FOR j: = 2 TO n DO triangolo [1,j]: = 0;
   writeln (triangolo [1,1]);
   FOR i: = 2 TO n DO
      BEGIN
         triangolo [i, 1]: = 1;
         write (triangolo [i,1], '   ');
         FOR j: = 2 TO i DO
            BEGIN
               triangolo [i,j]: = triangolo [i - 1,j] +
                                  triangolo [i - 1,j - 1];
               write (triangolo [i,j], '   ')
            END;
         FOR j: = i + 1 TO n DO triangolo [i,j]: = 0;
         writeln
      END;
   END.
```

In seguito all'esecuzione avremo:

```
1
1   1
1   2    1
1   3    3    1
1   4    6    4    1
1   5   10   10    5    1
1   6   15   20   15    6    1
1   7   21   35   35   21    7    1
1   8   28   56   70   56   28    8    1
1   9   36   84  126  126   84   36    9    1
```

1.5 - I vettori matematici

I vettori ad una dimensione possono anche servire a rappresentare il concetto di vettore matematico.

Infatti un vettore $\vec{V}$ matematico è caratterizzato dall'insieme delle sue coordinate:

oppure

$$\vec{V} = (x_1, x_2 \dots x_n)$$

Nella maggior parte dei linguaggi di programmazione questo vettore sarà definito con una matrice *X* ad una dimensione, il cui generico elemento è *x*(*i*).

Questo si può fare anche in Pascal, ma è preferibile definire, in maniera più specifica, un tipo *coordinata* ed un tipo *vettore matematico*. Ad esempio, possiamo dichiarare:

```
TIPO
   coordinata = (x1, x2, x3, x4, x5)
   vettormat = VETTORE [coordinata] DI reale
```

1.5.1 - Lettura delle coordinate di un vettore matematico

Supponiamo di avere il programma seguente:

```
PROGRAM vett;
TYPE
   coord = (x1, x2, x3);
   vettormat = ARRAY [coord] OF real;
VAR
   v: vettormat;
   c: coord;
BEGIN
   FOR c: = x1 TO x3 DO
      read (v[c]);
END.
```

v[*c*] rappresenta la coordinata *c* del vettore $\vec{V}$. Quindi, nel nostro esempio, leggeremo di seguito

v [x1], v [x2], v [x3]

Appare chiaro che il discorso è generalizzabile a vettori matematici di *n* coordinate.

I nomi delle variabili devono corrispondere all'insieme dei valori scalari definito nel tipo coordinata: questi valori potrebbero altrettanto bene essere *x*, *y* e *z*.

1.5.2 - Prodotto scalare dei vettori matematici

Il prodotto scalare di due vettori

$$\vec{V} = (x1,\ x2,\ \ldots\ xn)$$
$$\vec{W} = (y1,\ y2,\ \ldots\ yn)$$

è definito da

$$\vec{V} \times \vec{W} = x1 \cdot y1 + x2 \cdot y2 + \ldots + xn \cdot yn$$

Sappiamo che, se il prodotto scalare di due vettori è nullo, i vettori sono ortogonali, e sappiamo pure che il prodotto scalare di un vettore per se stesso:

$$\vec{V} \times \vec{V} = x1^2 + x2^2 + \ldots + xn^2$$

rappresenta il quadrato della lunghezza del vettore medesimo. Quindi la lunghezza di un vettore è:

$$|V| = \sqrt{x1^2 + x2^2 + \ldots + xn^2}$$

Nei programmi che seguono calcoleremo il prodotto scalare di due vettori e la lunghezza di un vettore.

```
PROGRAM prodottoscalarevettore;
TYPE
  coord = (x1, x2, x3);
  vettormat = ARRAY |coord| OF real;
VAR
  v, w: vettormat;
  pscal: real;
  c: coord;
BEGIN
  pscal: = 0;
  FOR c: = x1 TO x3 DO
  BEGIN
    read (w|c|); read (v|c|);
    pscal: = pscal + v|c| * w |c|
  END;
  writeln (pascal)
END.
```

Esecuzione

```
    1    2    3    1    2    2    Ⓡ
  □ 9
```

Il calcolo della lunghezza di un vettore è un caso particolare del programma precedente. Il programma corrispondente è:

```
PROGRAM lunghezzavettore;
uses transcend;
TYPE
   coord = (x1, x2, x3);
   vettormat = ARRAY [coord] OF real;
VAR
   v: vettormat;
   lunghezza: real;
   c: coord;
BEGIN
   lunghezza: = 0;
   FOR c: = x1 TO x3 DO
   BEGIN
      read (v[c]);
      lunghezza: = lunghezza + v[c] * v[c]
   END;
   lunghezza: = sqrt (lunghezza);
   writeln (lunghezza)
END.
```

Esecuzione

```
  2    1    3       Ⓡ
□ 3.74165
```

1.5.3 - Esempi di programmi sui vettori

1. Scrivere un programma che calcoli il prodotto vettoriale di due vettori.
   Il prodotto vettoriale di due vettori $\vec{X}$ ed $\vec{Y}$ è anch'esso un vettore, definito da

   $$\vec{Z} = \vec{X} \wedge \vec{Y}$$

   tale per cui, se

   $$\vec{X} \begin{cases} x1 \\ x2 \\ x3 \end{cases} \qquad \vec{Y} \begin{cases} y1 \\ y2 \\ y3 \end{cases}$$

avremo:

$$\vec{Z} \begin{cases} x2y3 - x3y2 \\ x1y3 - x3y1 \\ x1y2 - y1x2 \end{cases}$$

2. Scrivere un programma che calcoli il prodotto misto di tre vettori $\vec{X}$, $\vec{Y}$, $\vec{Z}$.
   Il prodotto misto è definito da

   $$pm = \vec{X} \times (\vec{Y} \wedge \vec{W})$$

   cioè è dato dal prodotto scalare del primo vettore per il prodotto vettoriale degli altri due.
   Si tratta di una grandezza scalare che misura algebricamente il volume del parallelepipedo costruito su questi tre vettori.

Soluzioni

1. Prodotto vettoriale.

```
PROGRAM prodvettoriale;
TYPE
   coord = (x1, x2, x3);
   vettormat = ARRAY |coord| OF real;
VAR
   u, v, w = vettormat;
   c: coord;
BEGIN
   FOR c: = x1 TO x3 DO
   BEGIN
      read (w|c|); read (v|c|);
   END;
   u|x1|: = v|x2| * w|x3| - v|x3| * w|x2|;
   u|x2|: = v|x1| * w|x3| - v|x3| * w |x1|;
   u|x3|: = v|x1| * w|x2| - v|x2| * w|(x1|;
FOR c: = x1 TO x3 DO writeln (u|c|, '   ')
END.
```

2. Prodotto misto.
   Il programma è, a questo punto, immediato.

1.6 - Altre strutture lineari: le pile e le code

La struttura del tipo vettore ad una dimensione permette di rappresentare anche strutture di dati in cui l'ordine d'ingresso e di uscita degli elementi è significativo.

Questo è il caso delle strutture a *pila*, il cui funzionamento è del tipo: *ultimo* arrivato *primo* uscito (*last* in *first* out). Una pila può venir simulata mediante un vettore ed un parametro, detto altezza, che indica la sommità della pila. I dati vengono introdotti in cima alla pila, e analogamente dalla cima vengono estratti i dati. Questa struttura si può simulare con il programma seguente:

```
PROGRAM pila;
CONST n = 10;
VAR pila: ARRAY [1 ... n] OF integer;
      altezza: 0 ... n;
      dato: integer;
BEGIN
   altezza: = 0;
   WHILE altezza < n DO
      BEGIN
         read (dato);
         altezza: = succ(altezza);
         pila [altezza]: = dato
      END;
   WHILE altezza > 0 DO
      BEGIN
         dato: = pila [altezza];
         altezza: = pred (altezza);
         write (dato, '   ')
      END;
END.
```

Esecuzione

```
3   5   1   6   8   3   7   9   2   0
0   2   9   7   3   8   6   1   5   3
```

Un'altra struttura possibile è la struttura a *coda*, o *fila d'attesa*, caratterizzata da un funzionamento del tipo: *primo* arrivato *primo* uscito (*first* in *first* out).

Questa struttura può essere simulata dal programma seguente:

```
PROGRAM filaattesa;
CONST n = 10;
VAR filaat: ARRAY [1 ... n] OF integer;
      altezza: 0 ... n;
      dato: integer;
BEGIN
   altezza: = 0;
   WHILE altezza < n DO
      BEGIN
         read (dato);
         altezza: = succ(altezza);
         filaat [altezza]: = dato
      END;
   altezza: = 0;
   WHILE altezza < n DO
      BEGIN
      altezza: + succ(altezza);
         dato: = filaat[altezza];
         write (dato, '   ')
      END;
END.
```

Esecuzione

```
3    5    1    6    8    3    7    9    2    0
3    5    1    6    8    3    7    9    2    0
```

AVVERTENZE

Nella pratica, i processi che accedono a queste strutture possono essere dei processi paralleli. Di questo riparleremo nel Capitolo 7, dedicato alle strutture dinamiche con puntatori.

2 - IL TRATTAMENTO DELLE MATRICI

Nel linguaggio Pascal non esistono istruzioni specifiche per il trattamento delle matrici. È comunque possibile considerare i vettori a due dimensioni come matrici, e quindi programmare le consuete operazioni matriciali. Considereremo per ora solo le matrici quadrate.

2.1 - Definizione di matrice unitaria

La matrice unitaria è caratterizzata dal fatto che gli elementi della diagonale principale sono uguali all'unità: ad esempio

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Una matrice siffatta è caratterizzata dagli elementi

$a_{ii} = 1$ per qualunque valore di i

e

$a_{ij} = 0$ per $i \neq j$

e può essere generata dal programma seguente:

```
PROGRAM matriceunitaria;
CONST
   nriga = 3;
   ncolon = 3;
TYPE
   indrig = 1 ... nriga;
   indcol = 1 ... ncolon;
   riga = ARRAY [indcol] OF real;
   matrice = ARRAY [indrig] OF riga;
VAR
   mat: matrice;
   i: indcol;
   j: indrig;
BEGIN
   FOR i: = 1 TO ncolon DO
      BEGIN
         FOR j: = 1 TO nriga DO
         BEGIN
            IF i = j THEN mat [i] [j]: = 1
                     ELSE mat [i] [j]: = 0;
            write (mat[i] [j])
         END;
         writeln;
      END;
END.
```

In questo programma la matrice è stata definita come un vettore di righe, ma avrebbe potuto essere definita anche come un vettore di colonne. Un generico elemento della matrice è espresso da *mat*[i] [j].

2.2 - Somma di due matrici

La somma di due matrici consiste nel sommare gli elementi con lo stesso indice. Ad esempio, la matrice *C(i,j)* = *A(i,j)* + *B(i,j)* è ottenuta dal programma seguente:

```
PROGRAM matricesomma;
CONST
  nriga = 3;
  ncolon = 3;
TYPE
indrig = 1 ... nriga;
indcol = 1 ... ncolon;
  matrice = ARRAY [indcol, indrig] OF real;
VAR
  mat1, mat2, mat3: matrice;
  i: indcol;
  j: indrig;
BEGIN
  FOR i: = 1 TO ncolon DO
    BEGIN
      FOR j: = 1 DO nriga DO
      BEGIN
        read (mat1[i,j]); read (mat2[i,j])
      END;
      writeln;
    END;
    FOR i: = 1 TO nriga DO
    BEGIN
      FOR j: = 1 TO ncolon DO
      BEGIN
        mat3[i,j]: = mat1[i,j] + mat2[i,j];
      END;
    END;
    FOR i: = 1 TO ncolon DO
      BEGIN
      FOR j: = 1 TO nriga DO
          write (mat3[i,j]);
          writeln
      END;
END.
```

In questo programma abbiamo utilizzato un vettore a due dimensioni, definito direttamente per mezzo di due indici, l'uno di riga e l'altro di colonna. Un elemento del vettore è quindi rappresentato mediante *mat[i,j]*.

La parte vera e propria di calcolo è molto breve. Le rimanenti istruzioni sono istruzioni di lettura delle matrici, e soprattutto istruzioni per la stampa linea per linea, con un'edizione ridotta al minimo. Il risultato ottenuto sarà pertanto, ad esempio:

$$\begin{matrix} 3 & 7 & 3 \\ 3 & 5 & 3 \\ 7 & 7 & 3 \end{matrix} = \begin{matrix} 1 & 3 & 2 \\ 0 & 5 & 1 \\ 3 & 2 & 3 \end{matrix} + \begin{matrix} 2 & 4 & 1 \\ 3 & 0 & 2 \\ 4 & 5 & 0 \end{matrix}$$

2.2.1 - Trasposizione di una matrice

La trasposizione di una matrice è ottenuta invertendo fra loro righe e colonne della matrice stessa. Se t_{ij} è l'elemento della matrice trasposta, avremo: $t_{ij} = a_{ji}$.

Il programma relativo è il seguente:

```
PROGRAM matricetrasposta;
CONST
  nriga = 3;
  ncolon = 3;
TYPE
  indrig = 1 ... nriga;
  indcol = 1 ... ncolon;
  matrice = ARRAY [indcol, indrig] OF real;
VAR
  mat, matt: matrice;
  i: indcol;
  j: indrig;
BEGIN
  FOR i: = 1 TO ncolon DO
    BEGIN
      FOR j: = 1 TO nriga DO
        read (mat[i, j]);
      writeln;
    END;
    FOR i: = 1 TO nriga DO
      BEGIN
        FOR j: = 1 TO ncolon DO
          matt[i,j]: = mat[j,i];
      END;
    FOR i: = 1 TO ncolon DO
      BEGIN
        FOR j: = 1 To nriga DO
          write (matt[i, j]);
          writeln
      END;
  END.
```

Esempio

Se

$$\text{mat} = \begin{pmatrix} 4 & 2 & 3 \\ 5 & 6 & 7 \\ 2 & 1 & 4 \end{pmatrix}$$

la matrice trasposta è:

$$\text{matt} = \begin{pmatrix} 4 & 5 & 2 \\ 2 & 6 & 1 \\ 3 & 7 & 4 \end{pmatrix}$$

Nelle matrici rettangolari, se le dimensioni della matrice iniziale sono (n,m), le dimensioni della matrice trasposta saranno (m,n).

2.3 - Moltiplicazione di una matrice per uno scalare

Data una matrice $A(n,n)$, otterremo una matrice $B(n,n) = k \cdot A\ (n,n)$ se moltiplicheremo tutti gli elementi di A per un coefficiente scalare k. In altre parole,

$$b_{ij} = k \times a_{ij}$$

Il programma che permette di eseguire questo calcolo è il seguente:

```
PROGRAM matricescalare;
CONST
   nriga = 3;
   ncolon = 3;
TYPE
   indrig = 1 ... nriga;
   indcol = 1 ... ncolon;
   riga = ARRAY[indcol] OF real;
   matrice = ARRAY[indrig] OF riga;
VAR
   mat: matrice;
   i: indcol;
   j: inrig;
   scalare: real;
BEGIN
   FOR i: = 1 TO nriga DO
      BEGIN
         FOR j: = 1 TO ncolon DO
```

```
            BEGIN
               read (mat[i] [j]
            END;
            writeln;
         END;
         write ('scalare ='); read (scalare);
         FOR i: = 1 TO nriga DO
         BEGIN
            FOR j: = 1 TO ncolon DO
            BEGIN
               mat [i] [j]: = scalare * mat[i] [j];
               write (mat[i] [j])
            END;
            writeln
         END;
      END.
```

2.4 - Moltiplicazione di due matrici

L'operazione di moltiplicazione è un po' più complessa, perché un elemento della matrice-prodotto è dato dalla somma dei prodotti fra gli elementi della riga *i* di una matrice e gli elementi della colonna *j* dell'altra matrice. Abbiamo cioè:

$$c_{ij} = \sum_{k-1}^{n} a_{ik} b_{kj}$$

Ed ecco un programma di moltiplicazione di due matrici:

```
PROGRAM matricemoltiplicazione;
   CONST
   nriga = 3;
   ncolon = 3;
TYPE
   indrig: 1 ... nriga;
   indcol: 1 ... ncolon;
   matrice = ARRAY[indcol, indrig) OF real;
VAR
   mat1, mat2, mat3: matrice;
   k, i: indcol;
   j: indrig;
BEGIN
   FOR i: = 1 TO ncolon DO
      BEGIN
         FOR j: = 1 TO nriga DO
```

```
        BEGIN
          read (mat1[i,j]); read (mat2[i,j])
        END;
        writeln;
      END;
      FOR i: = 1 TO nriga DO
      BEGIN
        FOR j: = 1 TO ncolon DO
        BEGIN
          mat3[i,j]: = 0;
          FOR k: = 1 TO ncolon DO
            mat3[i,j]: = mat3[i,j] + mat1 [i,k] * mat2[k,i];
        END;
      END;
      FOR i: = 1 TO ncolon DO
        BEGIN
          FOR j: = 1 TO nriga DO
            write (mat3[i,j]);
            writeln
        END;
  END.
```

In questo esempio, analogo a quello di somma, cambia solo la parte di calcolo: restano i due cicli su *i* e *j*, ma è stato opportuno aggiungere un terzo ciclo, che permette di calcolare l'elemento c_{ij} secondo la formula appena presentata.

Se introduciamo i dati

```
1   2   3   4   2   1   Ⓡ
0   3   5   0   1   2   Ⓡ
3   4   2   5   3   0   Ⓡ
```

all'esecuzione, con questi dati, avremo:

$$\begin{vmatrix} 19 & 14 & 7 \\ 19 & 5 & 10 \\ 24 & 27 & 7 \end{vmatrix} \doteq \begin{pmatrix} 1 & 3 & 2 \\ 0 & 5 & 1 \\ 3 & 2 & 3 \end{pmatrix} \times \begin{pmatrix} 2 & 4 & 1 \\ 3 & 0 & 2 \\ 4 & 5 & 0 \end{pmatrix}$$

OSSERVAZIONI

È possibile anche moltiplicare due matrici rettangolari, a patto che il numero delle colonne della prima matrice sia uguale al numero delle righe della seconda. Ad esempio, si può moltiplicare una matrice *A(l,n)* per una matrice *B(n,m)*, ottenendo una matrice-prodotto *C(l,m)*. Per fare questo, basta modificare il programma precedente cambiando i limiti delle istruzioni FOR (PER).

2.5 - Inversione di una matrice

Consideriamo un prodotto di matrici quadrate:

$$C = A \cdot B$$

Se C è la matrice-identità, che chiameremo I (per cui $I = A \cdot B$), allora possiamo dire che la matrice B è la matrice inversa di A, cioé

$$B = A^{-1}$$

Infatti, cosi come per i numeri razionali, l'inverso di una matrice è definito da A^{-1}, tale per cui

$$A \cdot A^{-1} = 1$$

Comunque esistono matrici che non hanno la relativa matrice inversa: si tratta di matrici che possiedono determinate proprietà (*determinante* uguale a 0). Nel prossimo capitolo presenteremo un programma di calcolo del determinante.

Esistono numerosi algoritmi d'inversione di una matrice; fra gli altri, il metodo della triangolazione per eliminazione. Il programma corrispondente è:

```
PROGRAM matriceinversione;
CONST
   nriga = 3;
   ncolon = 3;
TYPE
   indrig = 1 ... nriga;
   indcol = 1 ... ncolon;
   matrice = ARRAY [indcol, indrig] OF real;
VAR
   mat, mati: matrice;
   k, i: indcol;
   j: indrig;
   d: real;
BEGIN
   FOR i: = 1 TO ncolon DO
      BEGIN
         FOR j: = 1 TO nriga DO
            read (mat [i, j]);
         writeln;
      END;
      FOR i: = 1 TO nriga DO
```

```
        BEGIN
          d: = mat [i, i];
          mat [i, i]: = 1;
          FOR j: = 1 TO ncolon DO
            mat [i, j]: = mat [i, j]/d;
          FOR k: = 1 TO ncolon DO
            IF k < > i THEN
              BEGIN
                d: = mat [k, i];
                mat [k, i]: = 0;
                FOR j: = 1 TO ncolon DO
                  mat [k, j]: = mat [k, j] - mat [i, j] * d;
              END;
        END;
      FOR i: = 1 TO ncolon DO
        BEGIN
          FOR j: = 1 TO nriga DO
            write (mat [i, j]);
            writeln
        END;
  END.
```

Esecuzione

Data la matrice

$$\begin{pmatrix} 1 & 2 & 1 \\ -1 & 3 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$

si ottiene il seguente risultato:

```
 6.25000E-1    -3.75000E-1    -1.25000E-1
 2.50000E-1     2.50000E-1    -2.50000E-1
-1.25000E-1    -1.25000E-1     6.25000E-1
```

2.5.1 - Un'applicazione delle matrici inverse

Ci limiteremo ad un semplice esempio di risoluzione di un sistema di equazioni a due incognite.

Si abbia:

$$3x + 4y = 10$$
$$x + 5y = 7$$

In forma matriciale scriveremo:

$$\begin{pmatrix} 3 & 4 \\ 1 & 5 \end{pmatrix} \quad \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 10 \\ 7 \end{pmatrix}$$

A questo punto poniamo

$$A = \begin{pmatrix} 3 & 4 \\ 1 & 5 \end{pmatrix} \qquad X = \begin{pmatrix} x \\ y \end{pmatrix} \qquad B = \begin{pmatrix} 10 \\ 7 \end{pmatrix}$$

Le matrici X e B sono matrici particolari di dimensioni (2, 1), che rappresentano dei vettori. Il sistema di equazioni può allora venir scritto nella forma $A \cdot X = B$.

Per risolvere questo sistema di equazioni senza ricorrere alle matrici, possiamo usare, ad esempio, il metodo della sostituzione. La seconda equazione può essere scritta come

$$x = 7 - 5y$$

Operando la sostituzione nella prima equazione, si avrà:

$$3\ (7 - 5y) + 4y = 10$$

donde

$$21 - 15y + 4y = 10$$

che dà $y = 1$ ed $x = 2$.

Usando la notazione matriciale, e supponendo che esista una matrice inversa A^{-1}, avremo:

$$X = A^{-1}B$$

La matrice inversa è per definizione tale per cui

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} 3 & 4 \\ 1 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

cioè

$$3a_{11} + a_{12} = 1 \qquad 3a_{21} + a_{22} = 0$$
$$4a_{11} + 5a_{12} = 0 \qquad 4a_{21} + 5a_{22} = 1$$

donde

$$a_{11} = 5/11 \qquad a_{21} = -1/11$$
$$a_{12} = -4/11 \qquad a_{22} = +3/11$$

cioè

$$A^{-1} = \begin{pmatrix} 5/11 & -4/11 \\ -1/11 & 3/11 \end{pmatrix}$$

Allora, essendo

$$X = A^{-1} \times B$$

si ha:

$$X = \begin{pmatrix} 5/11 & -4/11 \\ -1/11 & 3/11 \end{pmatrix} \quad x \quad \begin{pmatrix} 10 \\ 7 \end{pmatrix}$$

da cui

$$X = \begin{pmatrix} 50/11 & -28/11 \\ -10/11 & +21/11 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Pertanto la soluzione è proprio $x = 2$, $y = 1$.

3 - I VETTORI IMPACCATI

Abbiamo visto che gli elementi di un vettore sono memorizzati in parole consecutive di memoria.

Dal momento che, da un lato, la dimensione della parola di memoria varia secondo i sistemi e, dall'altro, anche il numero di bits (elementi binari) necessari per la rappresentazione di un elemento di un vettore varia a seconda del tipo dei dati, lo spazio di memoria occupato da un vettore non è sempre quello ottimale. A parte i tipi intero e reale, il cui formato è definito in modo preciso, gli altri tipi possono corrispondere, su un dato sistema, a codici la cui lunghezza in bits è molto inferiore alla dimensione della parola usata su quel sistema. Ne risulta uno spreco di memoria, in particolare nel caso dei dati non numerici, quali stringhe di caratteri, booleani, etc.

Per ovviare a quest'inconveniente, il linguaggio Pascal dispone di una clausola che permette di dichiarare un vettore *impaccato*. Questo si traduce nel fatto che lo spazio occupato da ciascun elemento è quello *ottimale*, in rapporto al codice associato al tipo di vettore.

Se avessimo dichiarato, ad esempio,

```
VAR testo: ARRAY [1 ... 2000] OF char;
```

il vettore di caratteri *testo* occuperebbe duemila parole-memoria, qualunque sia la dimensione della parola. Se invece scriviamo:

```
VAR testo: PACKED ARRAY [1 ... 2000] OF char;
```

il vettore *testo* occuperà solo duemila bytes, perché per la codifica di un carattere occorrono otto bits (v. Appendice 1).

Pertanto in questo modo possiamo avere stringhe di caratteri contigui in memoria, pur rimanendo entro il Pascal standard. Abbiamo visto che nella versione dell'U.C.S.D. esiste un tipo standard, detto *string* (stringa di caratteri): questo tipo è equivalente ad un vettore impaccato.

Un vettore impaccato può essere usato nello stesso modo di un vettore non impaccato: in particolare, l'elemento *testo* [*i*] rappresenta il medesimo carattere nei due esempi precedenti. Segnaliamo tuttavia che alcune realizzazioni del Pascal non accettano l'elemento di un vettore impaccato come parametro di una procedura o di una funzione.

Il risparmio di spazio in memoria, nel caso dei vettori impaccati, ha come contropartita un calo di efficacia nel trattamento dei vettori, perché comporta l'uso di procedure di disimpaccamento (*unpack*).

Pertanto il Pascal standard raccomanda di effettuare in una volta sola le operazioni di impaccamento (*pack*) e di disimpaccamento (*unpack*) per l'intero vettore, piuttosto che elemento per elemento.

Queste procedure permettono di passare da un vettore impaccato ad un vettore disimpaccato, e viceversa.

Consideriamo i seguenti vettori *C* e *D*:

```
VAR
   c: PACKED ARRAY [i ... j] OF tipot;
   d: ARRAY [n ... m] OF tipot;
```

ove $m - n \geq i - j$.

La procedura di disimpaccamento *unpack* (*c, d, n*) è equivalente all'istruzione

```
PER I: = i A j FARE
   d [I - i + n]: = c [I];
```

Dunque gli elementi da *c* [*i*] fino a *c* [*j*] vengono disimpaccati negli elementi da *d* [*n*] fino a *d* [$j - i + n$].

C'è poi la procedura d'impaccamento *pack (d, n, c)*, che equivale all'istruzione

```
PER I: = i A j FARE
   c [I]: = d [I - i + n]
```

Qui gli elementi da *d* [*n*] fino a *d* [$j - i + n$] sono impaccati negli elementi da *c* [*i*] fino a *c* [*j*] del vettore *C*.

L'istruzione di assegnazione può sussistere solo fra gli elementi di vettori dello

stesso tipo. Gli operatori relazionali sono ammessi fra gli elementi di vettori impaccati dello stesso tipo.

3.1 - Estensioni

Alcuni sistemi, e in particolare il Pascal dell'U.C.S.D., permettono di effettuare l'impaccamento di vettori di tipo qualsiasi. Ad esempio, con riferimento al vettore di booleani

```
b: PACKED ARRAY [0 ... 79] OF boolean;
```

dato che ogni elemento occupa un solo bit, l'intero vettore occuperà soltanto dieci bytes (80 bits).

Analogamente per i tipi sottocampo d'interi:

```
a: PACKED ARRAY [1 ... 8] OF [0 ... 15];
```

Qui, per rappresentare un elemento dell'insieme degli interi che vanno da 0 a 15, occorreranno quattro bits; quindi il vettore *A* occuperà quattro bytes. Se l'intervallo del sottoinsieme è troppo grande (più di 8 bits), non si avrà impaccamento:

```
b: PACKED ARRAY [1 ... 10] OF [0 ... 999];
```

In questo caso il vettore occupa dieci parole di sedici bits.

È anche possibile impaccare vettori a più dimensioni:

```
m: PACKED ARRAY [0 ... 79,0 ... 79] OF boolean;
```

M è una matrice booleana di ottanta righe ed ottanta colonne, che occupa pertanto solo ottocento bytes.

È invece impossibile impaccare elementi che di per sé occupano due (o più) parole contigue. Inoltre, se il numero di bits necessario per rappresentare un elemento non è un divisore di 16, i bits di peso più alto resteranno inutilizzati.

Segnaliamo che, quando si ha un vettore multidimensionale, la parola riservata PACKED dev'essere specificata davanti ad ogni dichiarazione di vettore, o, almeno, davanti a quella relativa all'ultima dimensione: in caso contrario, il vettore sarà considerato come non impaccato. Ad esempio, se scriviamo:

```
a: PACKED ARRAY [0 ... 10] OF ARRAY [0 ... 5] OF char;
```

il vettore non sarà impaccato.

Avremmo invece dovuto scrivere

```
a: PACKED ARRAY [0 ... 10] OF PACKED ARRAY [0 ... 5] OF char;
```

oppure

```
a: ARRAY [0 ... 10] OF PACKED ARRAY [0 ... 5] OF char;
```

o ancora

```
a: PACKED ARRAY [0 ... 10,0 ... 5] OF char;
```

3.2 - Campo di variazione degl'indici

Esiste una limitazione sugl'indici: essi devono corrispondere ad un risultato della funzione *ord (indice)*, che dev'essere un intero, positivo o nullo. Inoltre il loro valore massimo non deve superare la dimensione definita, né il valore di memoria disponibile!

4 - GL'INSIEMI

Il concetto di insieme, di uso comune in matematica, è disponibile anche nel linguaggio Pascal. Unica limitazione, in Pascal, come in tutte le realizzazioni informatiche, non esiste il concetto di insieme infinito, perché il supporto di memoria è necessariamente finito. Un insieme può essere definito relativamente ad elementi appartenenti al tipo *scalare* o al tipo *sottocampo di scalare*. È anche possibile definire insiemi i cui elementi siano anch'essi definiti da un'*espressione*. Quindi un insieme può venir definito per enumerazione. Esiste poi un insieme nullo o vuoto, che non contiene nessun elemento.

Esempi

1) [ ] in Pascal rappresenta l'insieme vuoto, espresso in matematica con Ø.
2) ['a', 'b', 'c', 'd', 'e'] rappresenta l'insieme delle lettere da *a* ed *e*.
3) [3 ... 15] rappresenta l'insieme dei numeri interi da 3 a 15.
4) [x + y, x * y] rappresenta l'insieme dei valori delle due espressioni.
5) [gennaio ... dicembre] rappresenta l'insieme dei *mesi*.
6) [pera, mela, ciliegia, fragola, banana, arancia] è un insieme in cui sono elencati tutti gli elementi.

4.1 - Insieme base e tipo set (insieme)

In Pascal il concetto di insieme è riferito ad un insieme di oggetti dello stesso tipo scalare. È quello che chiamiamo *insieme base*, partendo dal quale si possono definire i suoi sottoinsiemi: questo è il *tipo set* (*insieme*).

Ad esempio, per il tipo scalare

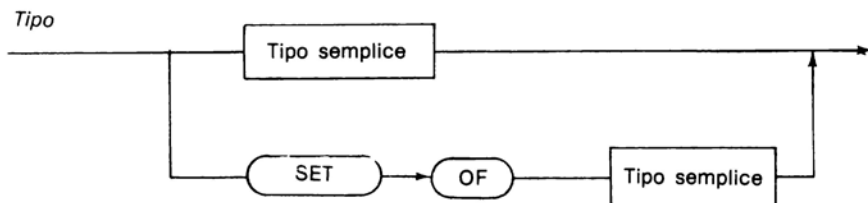
```
colorebase = (azzurro, rosso, giallo)
```

possiamo definire il tipo insieme *colore*, definito dai seguenti sottoinsiemi:

– ad un elemento: [azzurro] [rosso] [giallo]
– a due elementi: [azzurro, rosso] [azzurro, giallo] [rosso, giallo]
– a tre elementi: [azzurro, rosso, giallo]
ai quali bisogna aggiungere l'insieme vuoto [ ].

Il tipo *colore* può quindi assumere otto valori, ovvero 2^3 valori.
Come regola generale, se l'insieme base può assumere n valori, il tipo insieme potrà assumere 2^n valori.

4.2 - La definizione del tipo set (insieme)

La definizione del tipo SET (INSIEME) ha luogo generalmente in una dichiarazione di tipo, ma può anche avvenire direttamente nella definizione del tipo di una variabile, come si è già visto per tutti i tipi semplici.
Questa dichiarazione utilizza le parole riservate SET OF (INSIEME DI). Il diagramma sintattico è il seguente:

Esempio

```
TYPE
  sintomi = (febbre, eritema, nausea, tosse, vomito, tachicardia, anoressia);
  sindrome = SET OF sintomi;
```

Qui è stato definito un tipo insieme *sindrome* come un insieme di *sintomi.*

Dal diagramma sintattico vediamo che non è possibile definire un tipo insieme da un tipo che è anch'esso un tipo insieme.
I membri, o elementi, di un tipo insieme sono rappresentati da valori dell'insieme base corrispondente, posti fra parentesi quadre ([ ]).
Questi valori possono essere associati a variabili il cui tipo è un insieme.

Esempio

```
TYPE
  nota = (do, re, mi, fa, sol, la, si);
  suono = SET OF nota;
VAR
  accordo: suono;
```

Qui *nota* rappresenta l'insieme base. Il tipo *suono* è definito come un insieme di note (sottoinsieme dell'insieme base).

Si può poi definire una variabile *accordo*, i cui valori sono di tipo *suono*.

4.3 - Le operazioni sugl'insiemi

Nel caso di una procedura o di un programma, gl'insiemi sono espressi, fra parentesi quadre, da valori di tipo scalare o sottocampo. Ad esempio,

```
[azzurro ... rosso]
[si, no]
[do ... mi, sol ... si]
[1 ... 5, 20 ... 25, 50... 60]
```

4.3.1 - Unione di due insiemi

L'operazione di unione di due insiemi *A* e *B* consiste nel considerare tutti gli elementi, comuni o no ai due insiemi.

In matematica quest'operazione si esprime con il simbolo $\cup$:

$$A \cup B$$

In Pascal è espressa dal segno +, il che non genera ambiguità, perchè il tipo delle variabili esprime il tipo insieme.

Esempi

1) Se

```
A = [gatto, cane]
B = [pecora, mucca]
A + B = [gatto, cane, pecora, mucca]
```

2) Se

```
A = [0 ... 12]
B = [10 ... 18]
A + B = [0 ... 18]
```

3) Se

A = ['a'... 'd']
B = ['l' ... 't']
A + B = ['a'... 'd', 'l' ... 't']

4.3.2 - Intersezione di due insiemi

Quest'operazione permette di ottenere l'insieme degli elementi comuni ai due insiemi. In matematica è espressa con $A \cap B$; in Pascal con *.

Esempi

1) Se
A = [lunedì ... domenica]
B = [sabato, domenica]
A * B = [sabato, domenica]

2) Se
A = [do, re, mi]
B = [sol ... si]
A * B = [ ] insieme vuoto

3) Se
A = [0 ... 14]
B = [12 ... 25]
A * B = [12... 14]

4.3.3 - Differenza di due insiemi

La differenza di due insiemi *A* e *B* è definita dall'insieme degli elementi del primo insieme (*A*) che non compaiono nel secondo insieme (*B*).
In Pascal l'operazione si esprime con il segno –.

Esempio

Se
A = [Giovanni, Pietro, Michele]
B = [Giovanni, Alberto, Giorgio]
A – B = [Pietro, Michele]
B – A = [Alberto, Giorgio]

4.3.4 - Le relazioni fra insiemi

Fra due insiemi si possono usare anche alcuni operatori relazionati.
a) *Uguaglianza di due insiemi*

Si esprime con il segno =. Ad esempio,

[Carlo, Bianchi] = [Bianchi, Carlo]

b) *Disuguaglianza di due insiemi*
È espressa dall'operatore < >. Ad esempio,

[vecchio amico]< > [amico, vecchio]

c) *Inclusione di un insieme in un altro*
Un insieme *A* può essere contenuto, o incluso, in un altro insieme, *B*, se tutti gli elementi di *A* compaiono anche in *B*. Questa operazione si esprime con l'operatore =<.

Esempio

[do, re] =< [do ... si]

La relazione inversa è espressa dall'operatore >=, che indica che il primo insieme contiene il secondo.

Esempio

[1 ... 50] >= [2 ... 10]

Gli operatori relazionali puri e semplici < e > non sono utilizzabili con gl'insiemi. Per contro tutte le operazioni relazionali relative agl'insiemi possono essere espresse con valori booleani.

4.3.5 - L'operatore di appartenenza

In matematica, per esprimere che un elemento appartiene ad un insieme, si usa il simbolo $\in$; in Pascal si usa la parola riservata IN (IN).

Esempi

c IN [a, b, c, d]

maggio IN [gennaio ... giugno]

Come regola generale, l'operatore di appartenenza (IN) si usa con due operandi, il primo dei quali è un'espressione di cui si vuol sapere se il suo valore scalare appartiene al secondo operando, che è di tipo insieme. Questo permette di esprimere una relazione vera o falsa, che assume pertanto un valore booleano.

L'operatore IN è un operatore di relazione che si colloca nel diagramma sintattico degli operatori relazionali nel modo seguente:

4.4 - Assegnazione delle variabili di tipo insieme

L'istruzione di assegnazione vale anche per le variabili di tipo insieme.
Ad esempio, possiamo scrivere

```
lega: = [rame, ferro] + [nichel]
```

e, analogamente,

```
weekend: = [sabato, domenica]
ponte: = [venerdì]
feriale: = [giovedì]
vacanza: = [feriale] + [ponte] + [weekend]
```

4.5 - Numero degli elementi di un insieme

Il Pascal standard non prevede una funzione standard che permetta di sapere il numero di elementi di un insieme, cioè il *cardinale* dell'insieme.

Questa funzione esiste però in alcune realizzazioni del Pascal; allora, se *I* = [1, 3, 5, 7], si ha: *card* (*I*) = 4.

In assenza di questa funzione, si può ottenere lo stesso risultato testando gli elementi di tipo insieme contenuti nell'insieme del quale si vuol sapere il numero di elementi.

Esempio

```
TYPE
  verdura= [carota, porro, patata, pisello, fagiolo, rapa, cavolo];
VAR
  card: integer;
  macedonia: SET OF verdura;
  v: verdura;
BEGIN
  macedonia: = [carota, pisello, fagiolo, patata];
  card: = 0;
  FOR v: = carota TO cavolo DO
    BEGIN
      IF v IN macedonia THEN
        card: = card + 1;
    END;
  write (card);
END.
```

4.6 - Costituzione di un insieme

In un programma, un insieme può essere definito soltanto tramite assegnazione. Per contro gli elementi di un insieme possono venir letti da istruzioni d'ingresso/uscita.

Esempio

```
TYPE
  lettere = ['a' ...'z'];
VAR
  car: char;
  parola: SET OF lettere;
BEGIN
  parola: = [ ];
  WHILE NOT eoln DO
  BEGIN
    read (car);
    parola: = parola + [car]
  END;
END.
```

Questa proprietà è applicabile anche agl'insiemi con elementi di tipo scalare.

AVVERTENZA

L'operazione di unione di un insieme di caratteri con un nuovo carattere non è assimilabile ad un'operazione di concatenazione: se lo stesso carattere viene introdotto due volte, l'insieme non ne è modificato.

4.7 - I vettori di insiemi

Nella definizione sintattica delle dichiarazioni di vettore niente impedisce di usare un tipo insieme.
Quindi possiamo senz'altro scrivere:

```
TYPE
  colore = (fiori, quadri, cuori, picche);
  onore = (re, dama, fante);
  figura = SET OF onore;
  gioco = ARRAY [colore] OF figura;
VAR
  mano: gioco;
```

Queste dichiarazioni definiscono una variabile *mano* come un vettore di indice *colore*, il cui tipo è un insieme di figure delle carte.

Possiamo allora avere delle assegnazioni di questo tipo:

```
mano [cuori]: = [dama, fante];
mano [picche]: = [re];
```

ma anche

```
mano [quadri]: = mano [cuori] + mano [picche];
```

Questa è un'unione d'insiemi equivalente in questo caso a

```
mano [quadri]: = [re, dama, fante];
```

4.8 - Applicazioni degl'insiemi

Ricerca dei numeri primi, fino ad *n*, con il metodo del setaccio di Eratostene.

Si tratta di un algoritmo molto semplice: si parte dall'insieme dei numeri interi, fino ad *n*. Ogni volta che si trova un numero primo, si eliminano dall'insieme tutti i suoi multipli (di qui il nome di setaccio).

Avremo allora il programma seguente:

```
PROGRAM Eratostene;
CONST
  n = 100;
TYPE
  intero = 1 ... 100;
VAR
  primo, setaccio: SET OF intero;
  numero: intero;
  i: integer;
BEGIN
  primo: = [ ];
  setaccio: = [2 ... n];
  numero: = 2;
  REPEAT
    WHILE NOT (numero IN setaccio) DO
      numero: = numero + 1;
    primo: = primo + [numero];
    write (numero, ' ');
    i: = numero;
    WHILE i <= n DO
      BEGIN
        setaccio: = setaccio – [i];
        i: = i + numero;
      END;
  UNTIL setaccio = [ ];
END.
```

Esecuzione

```
  2   3   5   7  11  13  17  19  23  29  31  37  41  43  47
 53  59  61  67  71  79  83  89  97
```

Il programma è un esempio tipico di uso degl'insiemi.

Questa formulazione ha il pregio di essere facile da scrivere e da capire, ma con tutto ciò non è la migliore (v. Wirth).

Il programma può essere modificato in modo da trovare una soluzione più economica come spazio di memoria: a tal fine ricorreremo ad un vettore impaccato di booleani, ottenendo il programma seguente:

```
PROGRAM Eratostene;
CONST
  n = 10000;
TYPE
  intero = 1 ... n;
VAR
  setaccio: PACKED ARRAY [intero] OF boolean;
  numero: intero;
  nr, i: integer;
BEGIN
  FOR i: = 1 TO n DO setaccio [i]: = true;
  numero: = 2;
  nr: = n − 1;
  REPEAT
    WHILE NOT (setaccio [numero]) DO
      numero: = succ (numero);
    write (numero, ' ');
    i: = numero;
    WHILE i <= n DO
      BEGIN
        IF setaccio [i] THEN
          BEGIN
            setaccio [i]: = false;
              nr: = nr − 1;
          END;
        i: = i + numero;
      END;
    writeln (nr);
  UNTIL nr = 0;
END.
```

Questa seconda versione offre il vantaggio di risparmiare spazio, e quindi di tollerare un setaccio molto più grande.

Per contro il tempo di calcolo sarà maggiore, perchè, ad ogni riferimento al vettore, occorrerà passare attraverso un'operazione di disimpaccamento per accedere al bit che rappresenta il valore booleano. Questo programma è pertanto un valido test delle prestazioni di un sistema Pascal. L'uscita dei primi numeri è quella che richiede tempi più lunghi; poi il setaccio si svuota ed i numeri escono con il ritmo del dispositivo di uscita usato dal sistema.

ESERCIZI

1. Divisione di un polinomio per $x - r$.
   Data l'equazione polinomiale
   $$a_n x^n + a_{n-1} x^{n-1} + \dots + a_0 = 0$$
   se r è una radice, è possibile dividere il polinomio per $r - x$.
   Si ottiene così il polinomio di grado $n - 1$.
   $$a_n x^{n-1} (a_{n-1} + ra_n) x^{n-2} + (a_{n-2} + (r(a_{n-1} + ra_n))x^{n-3} + \dots$$
   che ha la forma:
   $$c_k x^k + c_{k-1} x^{k-1} + \dots + c_1 x + c_0 = 0 \quad \text{con } k = n - 1$$
   ove i coefficienti c_i sono dati da:
   $$c_k = a_n$$
   $$c_{k-1} = (a_{n-1} + ra_n) = a_{n-1} + rc_k$$
   $$c_{k-2} = (a_{n-2} + r(a_{n-1} + ra_n)) = a_{n-2} + rc_{k-1}$$
   ....
   $$c_i = a_{i+1} + rc_{i+1}$$

2. Calcolo del valore di un polinomio mediante il metodo di Horner.
   Sapendo che un polinomio $p_n(x)$ di grado n può essere scritto come segue:
   $$p_n(x) = (\dots((a_0 x + a_1) x + a_2) x + \dots) x + a_n$$
   è possibile calcolare i coefficienti b_i, per i che varia da 0 ad n, riferendosi alle seguenti relazioni:
   $$b_0 = a_0$$
   $$b_1 = b_0 x + a_1$$
   ....
   $$b_n = b_{n-1} x + a_n$$
   Si ottiene così il valore del polinomio $p_n(x)$, cioè b_n.

3. Applicando il programma presentato in 2-5, calcolare la matrice inversa di un sistema lineare a tre equazioni.

4. Scomposizione di una matrice simmetrica in una matrice triangolare con il metodo di Choleski.
   Sapendo che una matrice può venir scomposta nel prodotto di due matrici triangolari e che, in particolare, se la matrice è simmetrica, la scomposizione porta al prodotto di due matrici, di cui l'una è la trasposta dell'altra:

   $$A = M * M^t$$

   il metodo di Choleski permette di ricavare i termini della suddetta matrice.
   Se a_{ij} è il generico termine della matrice A, ed m_{ij} quello della matrice M, avremo:

   $$m_{11} = \sqrt{a_{11}}$$
   $$m_{1j} = a_{1j}/m_{11}$$
   $$m_{22} = \sqrt{a_{22} - m_{12} * m_{12}}$$
   $$m_{2j} = (a_{2j} - m_{12} * m_{1j})/m_{22}$$
   ....
   $$m_{kk} = \sqrt{a_{kk} - \sum_{i=1}^{k-1} m_{ik} * m_{ik}}$$
   $$m_{kj} = (a_{ki} - \sum_{i=1}^{k-1} m_{ik} * m_{ij})/m_{kk}$$
   $$m_{nn} = \sqrt{a_{nn} - m^2{}_{1n} \ldots - m^2{}_{n-1n}}$$

   Scrivere un programma in Pascal che permetta di calcolare i coefficienti.

5. Scrivere un programma di ordinamento alfabetico.

6. Scrivere un programma che permetta di memorizzare una partitura musicale nella forma seguente:
   le note vanno introdotte così come sono, e cioè come do, re, mi, fa, sol, la, si. L'ottava sarà espressa dal codice o, seguito da una cifra indicante il numero dell'ottava; le pause saranno espresse dalla lettera p. Il ritmo sarà indicato da un numero, diverso a seconda della nota: 1 per una semiminima, 2 per una minima, 4 per una semibreve, 0,5 per una croma, 0,25 per una semicroma, e così via. Le note puntate saranno espresse dalla frazione corrispondente alla durata della nota.

Costruire un vettore che rappresenti le frequenze delle note di una partitura.

7. Dati un intero n ed n punti di coordinate (x_i, y_i) in un sistema di assi cartesiani, scrivere un programma in Pascal che determini la lunghezza del segmento compreso fra i punti (x_j, y_j) e (x_K, y_K). Leggere n, j, k ed il vettore dei punti.

8. Un quadrato magico è un quadrato suddiviso in caselle nelle quali i numeri interi, partendo da 1, sono disposti in modo tale che le somme dei numeri di ciascuna riga, di ciascuna colonna e di ciascuna diagonale siano uguali. Ad esempio,

4	9	2
3	5	7
8	1	6

dà come somma 15.

Diversi algoritmi permettono di ottenere quadrati magici di ordine dispari, cioè quadrati in cui è dispari il numero degli elementi di ciascun lato. Il più semplice è il seguente:
- l'elemento posto sotto quello centrale è 1;
- gli elementi successivi (2, 3, 4 ...) occupano le caselle poste all'intersezione della riga sottostante e della colonna di destra;
- qualora si sia su un lato esterno del quadrato, si continua con il lato opposto seguendo la stessa regola;
- se una casella è già occupata, il numero è posto sulla stessa colonna, due righe più sotto.

Scrivere un programma in Pascal che generi un quadrato magico con questo algoritmo.

Esempio

11	24	7	20	3
4	12	25	8	16
17	5	13	21	9
10	18	1	14	22
23	6	19	2	5

9. Scrivere un programma che stampi una data nel seguente formato:

n, gg, mm, aa

dove

n va da 1 a 7 per lunedì, martedì, mercoledì, giovedì, venerdì, sabato, domenica

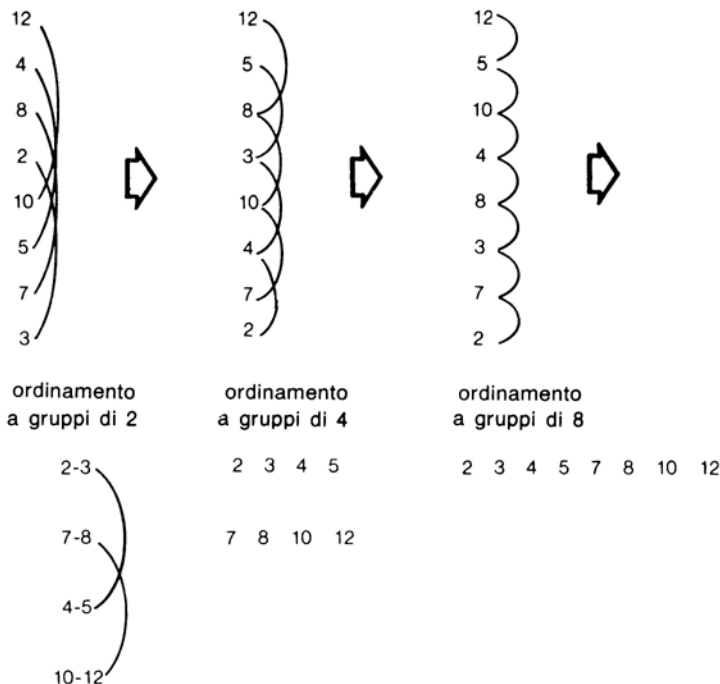
gg = giorno del mese

mm = numero d'ordine del mese

aa = anno

10. Modificare il programma dell'Esercizio 9 per trovare il giorno della settimana corrispondente ad una data compresa fra l'anno 1900 e l'anno 1999. Si terrà conto degli anni bisestili, e si assumerà che il 1° gennaio 1900 era un martedì.

11. Un altro algoritmo di ordinamento.
Illustriamo qui un altro algoritmo di ordinamento, detto algoritmo di Shell.
Il principio informatore è il seguente: dapprima si confrontano gli elementi a due a due nel modo seguente: $B(i)$ con $B(i + n/2)$, e si mettono in ordine. I gruppi di due elementi ordinati danno luogo, applicando il criterio precedente, a gruppi di quattro elementi ordinati; analogamente i gruppi di quattro elementi ordinati danno luogo a gruppi di otto, e così via fino ad avere un gruppo formato da tutti gli elementi disposti in ordine.

Esempio

Scrivere un programma in Pascal che usi quest'algoritmo.

12. Si abbia una rete di linee della metropolitana, alcune delle quali con una serie di coincidenze.
    Vogliamo scrivere un programma che, date due stazioni, *xi* ed *xj*, permetta di determinare il percorso ottimale.

    - Assumiamo inizialmente che il percorso ottimale è quello per cui vi è il minimo numero di stazioni fra *xi* ed *xj*; inoltre introduciamo per un cambio di linea una penalizzazione pari a cinque stazioni supplementari.
    - La rete di linee potrà venir espressa con un vettore a due dimensioni: *L* (i, j), ove *i* esprime un numero della linea e *j* il numero della stazione.
      Quindi, ad esempio, L (2, 12) esprimerà la dodicesima stazione della linea 2.
    - Una stazione sarà identificata da un numero di quattro cifre: *xxyy*, ove *xx* esprime il numero della linea, ed *yy* il numero della stazione.
    - Una stazione in cui non si ha coincidenza sarà indicata da un valore nullo in *L* (*i*, *j*); una stazione in cui si ha coincidenza conterrà il numero che specifica la stazione della linea incrociata.
      Ad esempio, *L* (*i*, *j*) = 504 esprimerà una coincidenza con la linea 5 alla stazione 4. Assumiamo che in una stazione possa esserci non più di una coincidenza.
    - Il programma dev'essere riferibile ad una qualsiasi rete metropolitana, e va corredata dallo schema della rete e da qualche esempio.

CAPITOLO 5

LE FUNZIONI E LE PROCEDURE

"L'immaginazione ha tutto: crea la bellezza, la giustizia, e la felicità, che è la cosa più importante del mondo...
Questi all'incirca sono gli effetti di questa facoltà ingannevole che sembra esserci data espressamente per indurci ad un errore necessario..."

PASCAL,
Pensées

Finora abbiamo esaminato le varie strutture delle istruzioni del linguaggio Pascal, senza far intervenire i concetti di procedura e di funzione. Infatti i programmi che abbiamo illustrato contenevano un unico blocco – il programma principale – che, eventualmente, richiamava funzioni o procedure standard del linguaggio, che non occorreva dichiarare.

In alcuni casi il programmatore può avere l'esigenza di definire delle funzioni particolari, che non esistono in versione standard. Analogamente, quando un programma supera un certo grado di complessità, può essere necessario scomporlo in moduli che eseguano un lavoro specifico, programmato una volta per tutte: ad esempio, la moltiplicazione di due matrici, un ordinamento, un algoritmo d'identificazione, etc. Infine, può capitare che la stessa elaborazione si ripeta più volte in un programma: in questo caso è utile poter richiamare più volte uno stesso modulo.

Ai lettori che conoscono altri linguaggi di programmazione questi concetti sono noti per lo più sotto il nome di sottoprogrammi.

Ma va precisato che in Pascal c'è una differenza fondamentale: queste funzioni, o sottoprogrammi, sono in grado di "richiamarsi da sole", e di conseguenza permettono di scrivere direttamente degli algoritmi ricorsivi.

Vedremo che, in alcuni casi, questo semplifica il lavoro del programmatore: i pro-

grammi così ottenuti sono facili da scrivere e da capire, anche se, d'altra parte, non sono sempre i più efficaci.

1 - LE FUNZIONI

Una funzione è un'entità che, dal punto di vista dell'utilizzatore o del programmatore, fornisce un unico risultato, a fronte di una serie di parametri d'ingresso.

La funzione si può esprimere nella forma *f* (*p1*, *p2* ... *pn*), in cui le *p* sono i parametri, o *argomenti* d'ingresso della funzione: quindi, per poterla calcolare, è necessario che, nel momento in cui essa viene richiamata, tutti i parametri siano già stati definiti.

Una funzione è caratterizzata da un'*intestazione*, che descrive il nome della funzione stessa ed i parametri da trasferire, seguita da un *corpo*, che rappresenta l'algoritmo che esegue le elaborazioni relative alla funzione.

Una funzione dev'essere *dichiarata* nella parte delle dichiarazioni del programma, e di conseguenza deve precedere il corpo del programma che ad essa potrebbe far riferimento.

Quindi il corpo di una funzione ha in sè tutte le sezioni di un programma: dichiarazioni e blocchi d'istruzioni eseguibili. Una funzione può a sua volta contenere delle dichiarazioni di funzioni, di procedure o di variabili interne.

Tutti gli oggetti definiti all'interno di una funzione sono detti *locali*: ad essi non è possibile accedere dal di fuori del corpo della funzione stessa. Invece gli oggetti definiti nella parte dichiarazione di un blocco di livello superiore sono detti *globali*: si può accedere ad essi da tutti i blocchi posti ai livelli inferiori.

In Pascal dobbiamo distinguere fra le funzioni standard e le funzioni dichiarate dal programmatore.

1.1 - Le funzioni standard

Esistono vari tipi di funzioni standard che saranno illustrati nei sottoparagrafi seguenti.

1.1.1 - Le funzioni matematiche

Queste funzioni si caratterizzano per il fatto di essere funzioni del tipo *f* (*x*), in cui vi è un solo argomento, di tipo intero o reale. Il risultato della funzione è anch'esso, secondo i casi, reale o intero. Queste funzioni non devono essere dichiarate, per cui sono utilizzabili direttamente in un programma.

Possono comparire solo al primo membro di un'istruzione di assegnazione, o nel richiamo di procedura o di funzione.

1.1.1.1 - La funzione valore assoluto: abs(x)

L'argomento *x* può essere un reale o un intero. Il risultato è reale o intero, a seconda del tipo dell'argomento.

Esempio

```
PROGRAM valass;
VAR
   x: real;
   i: integer;
BEGIN
   WHILE NOT eoln DO
      BEGIN
         read (i); write (abs(i) );
         read (x); write (abs(x) );
      END;
END.
```

Esecuzione

```
–10  10  –7.5  7.50000  5  5  5.431  5.43100
```

1.1.1.2 - La funzione elevazione al quadrato: sqr(x)

Anche qui l'argomento *x* può essere intero o reale; lo stesso vale per il risultato.

Esempio

```
PROGRAM superficie;
VAR
   c: char;
   r: real;
   l: integer;
   s: real;
BEGIN
   read (c);
   CASE c OF
      'r': BEGIN
           read (r);
           s: = 3.14 * sqr(r);
           writeln ('cerchio =', s);
           END;
      'l': BEGIN
           read (l);
           write ('quadrato =', sqr(l) );
           END;
   END;
END.
```

In questo programma si calcola la superficie di un cerchio, o quella di un quadrato, a seconda che s'introduca il carattere *r*, per il raggio, o il carattere *l*, per la lunghezza del lato.

1.1.1.3 - La funzione radice quadrata: sqrt(x)

Il parametro di questa funzione è di tipo reale o intero positivo; il risultato invece è di tipo reale.

La funzione è già stata considerata più volte nei capitoli precedenti. Bisogna solo accertarsi che l'argomento sia positivo, cosa che, in particolare, si ottiene considerando il valore assoluto dell'argomento:

```
sqrt(abs(x))
```

1.1.1.4 - Le funzioni trigonometriche

Sono le funzioni *sin*(*x*) e *cos*(*x*) e la funzione inversa *arctan*(*x*). La funzione *tg*(*x*) *non è* una funzione standard.

Per *cos(x)* e *sin(x)* gli argomenti, espressi in radianti, sono valori interi o reali. I risultati sono di tipo reale.

Il parametro può venir espresso sotto forma di una qualunque espressione aritmetica, il cui risultato è un numero esprimente un valore in radianti.

Per *arctan* l'argomento è intero o reale; il risultato, reale, è espresso in radianti.

A questo punto possiamo vedere alcuni problemi di trigonometria.

1) Conversione dei radianti in gradi, e viceversa.
   Sappiamo che $\pi = 180°$.
   Se il valore di un angolo è dato in gradi *g*, il suo valore *r* in radianti è

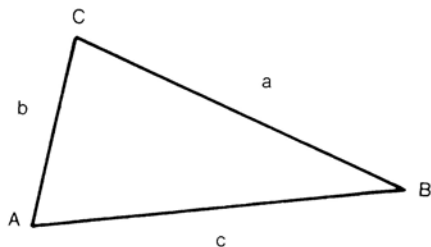
$$r = \frac{g}{180} \cdot \pi$$

   Inversamente, se è noto il valore in radianti, avremo:

$$g = \frac{r \cdot 180}{\pi}$$

   Come esercizio si suggerisce di scrivere un programma che calcoli i valori delle funzioni trigonometriche, sapendo che gli angoli sono dati in gradi e ricordando che la funzione tangente si ottiene calcolando

$$\text{tg } x = \frac{\text{sen } x}{\cos x}$$

2) Calcolo degli angoli di un triangolo.
   Si abbia un triangolo ABC, in cui le lunghezze dei lati siano *a*, *b*, *c*.

Gli angoli opposti ai rispettivi lati sono *A*, *B*, *C*.
Applicando il teorema di Pitagora sappiamo che

$$a^2 = b^2 + c^2 - 2bc \cos A$$

cioè

$$\cos A = \frac{b^2 + c^2 - a^2}{2bc}$$

Non disponiamo della funzione *arcoseno(A)*, e quindi dobbiamo ricorrere a delle relazioni trigonometriche semplici. In particolar modo,

$$\text{sen}^2 A + \cos^2 A = 1$$

cioè

$$\text{sen } A = \sqrt{1 - \cos^2 A}$$

e

$$\text{tg } A = \frac{\text{sen } A}{\cos A} = \frac{\sqrt{1 - \cos^2 A}}{\cos A}$$

da cui

$$A = \text{arcotangente } \left(\frac{\sqrt{1 - \cos^2 A}}{\cos A} \right)$$

Allo stesso modo, partendo da

$$b^2 = a^2 + c^2 - 2ac \cos B$$

avremmo:

$$\cos B = \frac{a^2 + c^2 - b^2}{2ac}$$

e

$$B = \text{arcotangente} \left(\frac{\sqrt{1 - \cos^2 B}}{\cos B} \right)$$

C si ottiene mediante la relazione

$$A + B + C = 180° \text{ (gradi)}$$

Attenzione: il problema non è sempre risolvibile, soprattutto se si danno ad *a*, *b* e *c* dei valori qualsiasi. Infatti, se

$$\frac{b^2 + c^2 - a^2}{2bc} > 1$$

l'angolo *A* non esiste; in altre parole, non esiste un triangolo corrispondente ai valori *a*, *b*, *c*. Il programma sarà dunque il seguente:

```
PROGRAM triangolo;
uses transcend;
CONST
  pi = 3.14159;
VAR
  a, b, c, ab, ac, bc: real;
FUNCTION coseno (ac, ab, bc: real): real;
BEGIN
  coseno: = (sqr(ac) + (sqr(ab) - sqr(bc))/(2 *ac* ab)
END;
FUNCTION seno (a, b, c: real): real;
BEGIN
  seno: = sqrt(1 - sqr(coseno(a, b, c)))
END;
FUNCTION angolo (a, b, c: real): real;
BEGIN
  angolo: = arctan (seno(a, b, c)/coseno(a, b, c)) *180/pi
END;
(*programma principale*)
BEGIN
  read (ab, ac, bc);
  a: = angolo(ac, ab, bc);
  b: = angolo(bc, ab, ac);
  c: = 180 - a - b;
  writeln ('angolo A =' a, 'angolo B =' b, 'angolo C =' c);
END.
```

? 2, 3, 4 Ⓡ

□ angolo A = 28.9550243 angolo B = 46.5674635
angolo C = 104.477512

? 3, 4, 5 Ⓡ

□ angolo A = 36.8698977 angolo B = 53.1301023
angolo C = 90
(È infatti un triangolo rettangolo, perché $a^2 + b^2 = c^2$.)

? 1, 1, 1.414 Ⓡ

□ angolo A = 45.0086517 angolo B = 45.0086519
angolo C = 89.9826968
(Infatti questa volta è un triangolo rettangolo isoscele, perché $1.414 \simeq \sqrt{2}$.)

1.1.1.5 - Le funzioni esponenziale (exp) e logaritmo neperiano (ln)

Le funzioni matematiche corrispondenti sono e^x ed il logaritmo neperiano $log_e(x) = ln\ x$.

Sono due funzioni inverse, perché $ln(e^x) = x$. La base è il numero $e = 2.71828...$

Gli argomenti di queste funzioni sono interi o reali; il risultato è reale.

Come parametro si può usare una qualsiasi espressione matematica; unica limitazione, per il logaritmo *ln* il valore dell'espressione dev'essere positivo.

Ad esempio, si voglia fare il calcolo delle rate di restituzione di un prestito *p*, al tasso d'interesse *i*, e per una durata di *n* periodi: il periodo è in rapporto con il tasso d'interesse, che può essere, secondo i casi, mensile o annuo.

Sappiamo che la formula che dà il tasso di rimborso è

$$r = p \cdot \frac{i}{(1 - (1 + i)^{-n})}$$

Avremo pertanto:

```
PROGRAM prestito;
VAR
  p, r, i: real;
  n: integer;
BEGIN
  write ('prestito ='); readln (p);
  write ('interesse ='); readln (i);
  write ('durata ='); readln (n);
  r: = p * i/(1 - exp(-n * ln(1 + i)));
  FOR i: = 1 TO n DO
  BEGIN
    writeln (p, r);
    p: = p - r;
  END;
END.
```

Per ottenere funzioni a base generica ci si riferirà alle seguenti formule di conversione:

a) *Esponenziale a base qualunque*
La funzione esponenziale può servire per calcolare funzioni del tipo a^x, facendo riferimento alla corrispondenza

$$a^x = e^{x \ln a}$$

b) *Logaritmo a base qualunque*
La funzione logaritmo neperiano è la funzione inversa dell'esponenziale, perché per definizione

$$\ln e^x = x \qquad \text{e} \qquad e^{\ln x} = x$$

In particolare, se consideriamo un logaritmo a base *a*, e cioè

$$\log_a (x)$$

abbiamo:

$$\log_a (x) = \log_a (e^{\ln x})$$
$$\log_a (x) = \log_a (e) \cdot \ln (x)$$

In particolare, se $x = a$, avremo:

$$\log_a (a) = 1 = \log_a (e) \cdot \ln (a)$$

$$\ln (a) = \frac{1}{\log_a (e)}$$

Quindi

$$\log_a (x) = \frac{\ln (x)}{\ln (a)}$$

c) *Logaritmo decimale*
In base a quello che abbiamo visto nel punto precedente, possiamo definire:

$$\log_{10} (x) = \ln x/\ln 10$$

Il programma che segue definisce le funzioni *loga* ed *expa* a base qualunque. Un po' più avanti vedremo la sintassi di queste definizioni.

```
PROGRAM funzione;
uses transcend;
VAR
   x, base, f: real;
   funz: char;
(*logaritmo a base qualunque*)
FUNCTION loga (base, x: real): real;
BEGIN
   loga: = ln(x)/ln(base)
END;
(*esponenziale a base qualunque*)
FUNCTION expa (base, x: real): real;
BEGIN
   expa: = exp (x * ln (base) )
END;
BEGIN
   readln (funz);
   WHILE funz IN ['L', 'E'] DO
   BEGIN
      readln (base);
      readln (x);
      CASE funz OF
         'L': f: = loga (base, x);
         'E': f: = expa (base, x)
      END;
      writeln (f);
      readln (funz);
   END;
END.
```

Esecuzione

```
L Ⓡ
8 Ⓡ
64 Ⓡ
□ 2.0000
E Ⓡ
2 Ⓡ
8 Ⓡ
□ 2.56000E2
L Ⓡ
10 Ⓡ
1000000 Ⓡ
□ 6.00000
```

E Ⓡ
10 Ⓡ
4 Ⓡ
☐ 1.00000E4
L Ⓡ
10 Ⓡ
3 Ⓡ
☐ 4.77121E–1
L Ⓡ
10 Ⓡ
☐3.01030E–1 Ⓡ

d) *Funzioni iperboliche*
Queste funzioni non esistono in versione standard.
È comunque possibile definire le funzioni iperboliche *senh*, *cosh* e *tgh* utilizzando la funzione esponenziale:

sinh = (exp (x) – exp (–x))/2

cosh = (exp (x) + exp (–x))/2

$$\text{tanh} = \frac{\text{sinh}}{\text{cosh}}$$

Vedremo più avanti che, in un programma, queste funzioni si possono definire una volta per tutte in forma simbolica.

1.1.2 - Le funzioni di troncamento e di arrotondamento

Esistono due funzioni che permettono di ricavare valori interi approssimati da argomenti reali.

La prima è la funzione di troncamento, definita da *trunc (x)*, con argomento reale, che permette di ricavare il valore intero immediatamente inferiore ad *x*.

Così, avendo

```
x: = 7.69;
write (trunc (x) );
```

il risultato sarà 7.

Analogamente, se *x*: = –3.4, *trunc (x)* darà il valore –4.

La seconda è la funzione di arrotondamento, definita da *round (x)*, che permette di ricavare il valore intero più prossimo all'argomento reale *x*.

Pertanto, se $x > 0$, *round (x) = trunc (x + 0.5)*; se $x < 0$, *round (x) = trunc (x – 0.5)*.

Allora, ad esempio,

x: = 9.78 darà per *round (x)* il valore 10.
x: = −5.4 darà per *round (x)* il valore −5.
x: = −2.75 darà per *round (x)* il valore −3.

1.1.2.1 - Alcune applicazioni

1) Per trovare tutti i numeri primi minori di 1000, basterà per ogni numero n verificare se tutti gl'interi, da 2 al valore della radice quadrata di n, sono divisori del numero stesso. Infatti, prima di 2 non troviamo numeri che dividano n, e i numeri maggiori di $\sqrt{n}$ sono necessariamente associati ad un fattore che si sarebbe già dovuto trovare come divisore di n, perché $\sqrt{n} \cdot \sqrt{n} = n$ e, se prendiamo un numero $x > \sqrt{n}$, l'altro fattore y dev'essere $< \sqrt{n}$.

Quindi, per determinare se un numero è un numero primo, basta verificare se, dividendo n per tutti i numeri da 2 a $\sqrt{n}$, si ha sempre resto, cioè n modulo i diverso da 0. Avremo allora il programma seguente:

```
PROGRAM primo;
uses transcend;
CONST
  max = 1000;
VAR
  k, i, j, n: integer;
BEGIN
  j: = 0;
  FOR n: = 1 TO max DO
    BEGIN
      k: = trunc (sqrt (n) );
      i: = 2;
      WHILE (n MOD i < > 0) AND (i < = k)
        DO i: = i + 1;
      IF i > k THEN
        BEGIN
          j: = j + 1;
          IF j MOD 10 = 0 THEN writeln;
          write (n, ' ');
        END;
    END;
END.
```

Esecuzione

```
  1    3    5    7   11   13   17   19   23   29
 31   37   41   43   47   53   59   61   67   71
 73   79   83   89   97  101  103  107  109  113
```

2) Calcolo di un numero arrotondato fino ad un certo numero di decimali.

Negli esempi finora presentati, i numeri decimali erano stampati con varie cifre decimali, per quanto questa precauzione sia superflua nella maggior parte dei casi.

Per arrotondare un numero ad *n* cifre decimali, basta moltiplicare il numero per 10^n: a questo punto si considera la sola parte intera, che è quindi divisa per 10^n. Ad esempio, si abbia il numero 145.7895678, da arrotondare a due cifre decimali.

Moltiplicandolo per $10^2 = 100$, otteniamo 14578.95678, in cui la parte intera è 14578. Dividendo questo numero per $10^2 = 100$, abbiamo come risultato 145.78.

Se invece vogliamo tener conto della successiva cifra decimale (in questo caso 9), possiamo aggiungere 0.5 al numero ottenuto dalla moltiplicazione per 10^2, ottenendo così 14579.45678. Allora il numero arrotondato al valore decimale superiore è 145.79. Se la cifra successiva fosse stata una cifra compresa fra 0 e 4, avremmo arrotondato l'ultima cifra al valore decimale inferiore.

Per concludere, si suggerisce di scrivere un programma che permetta di eseguire l'operazione descritta.

1.1.3 - La funzione ord

Questa funzione ha come argomento un valore *scalare* non reale.

Il risultato è un intero positivo o nullo, che rappresenta il numero d'ordine del valore scalare nel tipo scalare associato. Il primo valore ha numero d'ordine zero.

Esempio

Se

mese = (gennaio ... dicembre)

si ha:

ord (marzo) = 2

Ma attenzione: se

lettera = (b, c, d)

avremo

ord (c) = 1

Questo risultato è diverso da *ord ('c')*, che esprime il numero d'ordine del carattere *c* nell'insieme dei caratteri usati dal sistema.

1.1.4 - La funzione chr

Questa funzione permette di ottenere come risultato il carattere il cui numero d'ordine è specificato dall'argomento della funzione: *chr (n)* è tale per cui, se *car* è l'n-esimo carattere, ovvero se *car = chr (n)*, allora *n = ord (car).*

Perciò abbiamo:

    chr(ord(car)) = car

che mostra come, nel caso dei caratteri, le funzioni *ord* e *chr* siano l'una l'inverso dell'altra.

Esempio

```
PROGRAM seriedicaratteri;
VAR i: integer;
      car: char;
BEGIN
   FOR i: = 0 TO 255 DO
      BEGIN
         car: = chr(i);
         writeln (i, ' ', car)
      END;
END.
```

Questo programma permette di scrivere la serie dei caratteri del sistema utilizzato.

1.1.5 - Le funzioni predecessore e successore

Queste funzioni, definite anch'esse per argomenti di tipo scalare, permettono di ottenere il valore scalare che precede (*pred*), o che segue (*succ*) il valore dell'argomento nel tipo scalare associato. Si tratta di funzioni inverse, perchè

    succ (pred (x)) = x

Esempi

1) Nel tipo *mese* definito prima abbiamo:
   succ (maggio) = giugno

e

   pred (marzo) = febbraio

2) Le funzioni *succ* e *pred* si possono poi usare anche per incrementare o decrementare una variabile intera.

Se, ad esempio, abbiamo

```
VAR i: integer;
```

l'istruzione

```
i: = succ (i);
```

è equivalente a

```
        i: = i + 1;
```

Analogamente

```
        i: = pred (i);
```

è equivalente a

```
        i: = i − 1;
```

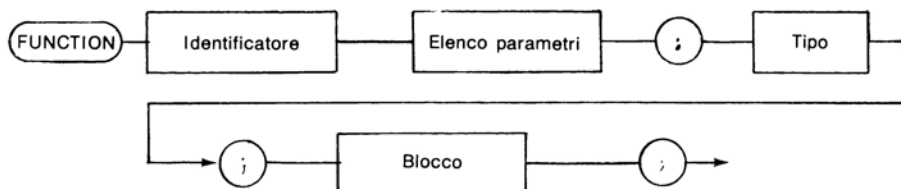
Attenzione: le funzioni *pred* e *succ* non sono definite per i valori estremi di una variabile scalare, perchè il primo elemento non ha alcun predecessore, e l'ultimo elemento non ha alcun successore.

1.2 - Dichiarazione delle funzioni

Si è visto che le funzioni standard sono caratterizzate dalla presenza di un identificatore e di un argomento, ovvero un parametro di tipo bene definito. Quando la funzione viene richiesta, si ottiene un risultato scalare unico, il cui tipo è anch'esso ben definito.

Le funzioni non standard definite dall'utilizzatore devono essere dichiarate nella sezione corrispondente. Questa dichiarazione è definita dall'*intestazione* della funzione, che comprende

- l'identificatore della funzione;
- il tipo del risultato;
- l'elenco dei parametri o argomenti, ed il loro tipo.

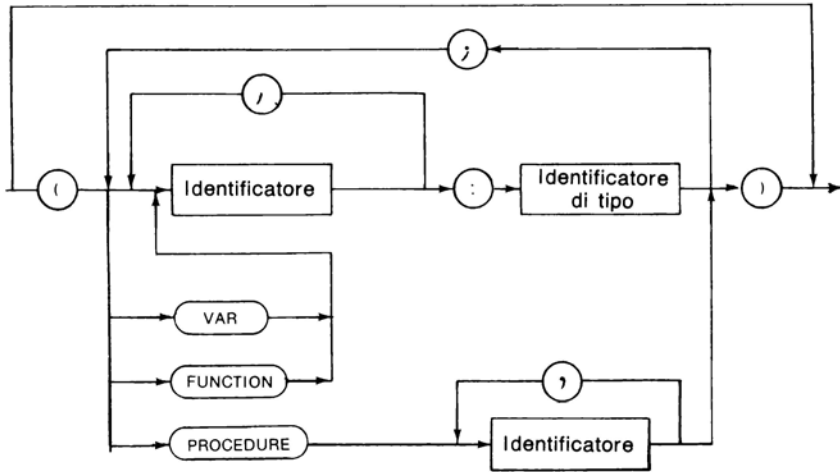
L'intestazione è a sua volta seguita da un blocco di programma. La sintassi relativa è data dal diagramma seguente:

La parola riservata FUNCTION (FUNZIONE) indica che siamo di fronte ad una dichiarazione di funzione. In uno stesso programma si possono avere diverse dichiarazioni di funzioni.

Il tipo è quello del risultato della funzione, e dev'essere di tipo scalare, sottocampo o puntatore (v. oltre).

1.2.1 - L'elenco dei parametri

L'elenco dei parametri d'ingresso può anche essere vuoto, ma in genere comprende almeno un argomento; in questo caso avremo una funzione ad una variabile x, e cioè $f(x)$, che abbiamo già incontrato a proposito delle funzioni standard. Comunque l'elenco può essere molto più complesso, e comprendere identificatori con il relativo tipo, variabili o funzioni sempre con il relativo tipo, o anche procedure.

L'elenco degli argomenti compare fra parentesi. Il diagramma sintattico è il seguente:

Elenco parametri

Da qui si vede che è possibile definire elenchi di parametri relativi a varie categorie (VAR, FUNCTION e PROCEDURE), separati da punti e virgola, e, all'interno di ciascuna categoria, più parametri dello stesso tipo, separati da virgole.

Il tipo dev'essere specificato da un identificatore, non da una dichiarazione.

1.2.2 - Il corpo della funzione

È costituito da un blocco di programma, e può contenere tutte le dichiarazioni e istruzioni del linguaggio. Comunque, come si è già detto, tutti gli oggetti definiti in tale blocco sono *locali*, per cui non sono accessibili dal di fuori della funzione. Bisogna quindi che il risultato della funzione sia trasmesso tramite l'identificatore della funzione stessa.

Nella sequenza d'istruzioni che costituisce una funzione, il nome (identificatore) della funzione comparirà al primo membro di un'istruzione di assegnazione; in caso contrario non si potrà accedere al risultato da nessun'altra parte del programma.

Generalmente, anche se non obbligatoriamente, tale istruzione di assegnazione è l'ultima del corpo della funzione.

Si tenga pertanto presente la seguente regola:

Il risultato di una funzione è unico, e dev'essere trasmesso mediante l'identificatore della funzione, il quale dovrà comparire a sua volta al primo membro di un'istruzione di assegnazione appartenente al corpo della funzione.

Esempio

Si debba definire la funzione trigonometrica

$$y = tg\ (x) = \frac{sen\ x}{cos\ x}$$

Il programma sarà il seguente:

```
PROGRAM funz;
VAR
  x, y: real;
FUNCTION tangente (z: real): real;
BEGIN
  tangente: = sin(z)/cos(z);
END;
BEGIN (*programma principale*)
  read (x);
  y = tangente (x)
  writeln (x, y);
END.
```

Qui il parametro *z* è trasmesso alla funzione sotto forma di un valore reale, che rappresenta l'angolo espresso in radianti: nella definizione di una funzione, è un parametro *muto*, che verrà sostituito da un valore noto *x* quando la funzione sarà richiesta. Il risultato è invece trasmesso nell'identificatore *tangente*.

Avremmo potuto definire la funzione anche in questo modo:

```
FUNCTION tangente (y: real): real;
BEGIN
  tangente: = sin (y)/cos (y);
END;
```

Qui *y* è sempre un parametro *formale*, o *muto*, che non ha niente a che vedere con la variabile *y* definita nel programma principale.

Quindi gl'identificatori utilizzati nell'elenco dei parametri possono avere nomi arbitrari, indipendenti dagl'identificatori usati al di fuori della funzione in questione. Tali

nomi servono semplicemente per individuare il ruolo dei parametri nell'algoritmo che calcola il risultato; nell'esecuzione della funzione, saranno sostituiti da parametri autentici, o *effettivi*.

1.2.3 - Funzioni come parametri

Volendo, con una sola funzione, definire per una variabile le funzioni trigonometriche cotangente e tangente, o, più generalmente, qualunque funzione caratterizzata da un quoziente di due funzioni, potremo scrivere:

```
PROGRAM funzionequoziente;
VAR tan, cot, x, y: real;
FUNCTION quoziente (FUNCTION n, d: real; x: real): real;
  BEGIN
    quoziente: = n(x)/d(x);
  END;
BEGIN
  readln (x);
  tan: = quoziente (sin, cos, x); write (tan);
  cot: = quoziente (cos, sin, x); write (cot);
  y: = quoziente (sin, sqr, x); write (y);
END.
```

Qui è stata definita una funzione *quoziente*, che permette di calcolare tangente e cotangente, ma anche una funzione quale $\frac{sen\ (x)}{x^2}$, o qualunque altra funzione *quoziente*.

Attenzione: la possibilità di utilizzare funzioni o procedure come parametri rientra nel Pascal standard; tuttavia non sempre è disponibile sui sistemi realizzati.

1.2.4 - Effetto margine e trasparenza delle funzioni

Sappiamo che, nel corpo di una funzione, è possibile riferirsi a variabili globali: questo non disturba in alcun modo, e può anzi essere utile purchè *non si modifichi* il valore di tali variabili. Invece l'impiego delle variabili globali al primo membro di un'istruzione di assegnazione, contenuta nel corpo di una funzione, provoca un *effetto margine* che deve assolutamente essere evitato, anche se ciò richiede particolari accorgimenti di programmazione.

Infatti una funzione (e vedremo che questo è vero anche per le procedure) dev'essere «trasparente»: deve cioè eseguire il calcolo per il quale è stata concepita senza modificare nient'altro oltre allo stato delle variabili locali.

Esempio

```
PROGRAM effettomargine;
VAR
  a, b, c: integer;
FUNCTION debole (z: integer): integer;
BEGIN
  a: = b + c;
  debole: = a + z;
END;
BEGIN
  a: = 1; b = 2; c = 3;
  y: = debole (a);
  writeln (a, b, c, y);
  b: = 1;
  y: = debole (b);
  writeln (a, b, c, y);
  c: = 1;
  y: = debole (c);
  writeln (a, b, c, y);
END.
```

Risultati

```
5   2   3   6
4   1   3   5
2   1   1   3
```

Nei tre casi la funzione *debole* dà un valore diverso per uno stesso valore (=1) del parametro. Le differenze sono dovute all'effetto margine ottenuto modificando la variabile *a* in un'istruzione di assegnazione.

1.2.5 - Trasmissione dei parametri

L'esempio precedente sottolinea la pericolosità dell'effetto margine quando si modificano i contenuti delle variabili globali nel corpo di una funzione.

Ci si può chiedere se si ha lo stesso effetto qualora un parametro della funzione corrisponda ad una variabile globale e se ne modifichi il valore nel corpo della funzione.

Esempio

Consideriamo il programma che calcola il M.C.D. mediante una funzione:

```
PROGRAM fmcd;
VAR
  i, j: integer;
FUNCTION mcd (k, l: integer): integer;
BEGIN
  WHILE k < > l DO
  IF k > l THEN k: = k – 1
           ELSE l = 1 – k;
  mcd: = 1
END;
BEGIN
  readln (i, j);
  writeln (mcd (i, j), i, j)
END.
```

Se $i = 18$ e $j = 24$, avremo in uscita

```
6    18    24
```

Di qui si vede che la funzione M.C.D. non ha modificato i valori di *i* e di *j*, trasmessi come parametri, benchè questi corrispondano alle variabili *k* ed *l*, modificate nel corpo della funzione. In questo caso i parametri sono trasmessi come *valori*: infatti non la variabile viene trasmessa, ma il suo valore.

Quindi in linea di principio i parametri devono venir trasmessi come valori, affinchè il risultato della funzione sia unico.

Comunque la sintassi dell'elenco dei parametri non impedisce di trasmettere delle variabili. Quindi, nell'esempio precedente, avremmo potuto definire la funzione in questo modo:

```
FUNCTION mcd (var k, l: integer): integer;
```

Il risultato allora sarebbe stato:

```
6    6    6
```

I parametri *i* e *j*, trasmessi come variabili, vengono modificati al ritorno della funzione.

Torneremo su questo punto studiando le procedure.

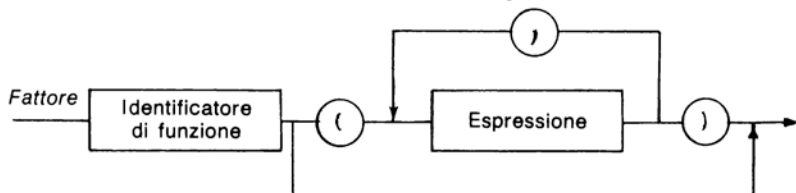
1.2.6 - Chiamata di una funzione e parametri effettivi

Dal punto di vista sintattico una funzione è considerata come un fattore; quindi, per richiederla, basta introdurla in un'espressione, come parametro o di un'altra funzione o di una procedura.

Nel momento in cui una funzione è chiamata tutti i parametri devono essere *effettivi*: vale a dire che devono essere già definiti.

L'ordine, il nome ed il tipo dei parametri effettivi devono corrispondere all'ordine ed al numero dei parametri formali, così come al loro *tipo*.

I parametri effettivi possono essere espressioni quando si ha la trasmissione per *valori*; invece, nel caso di funzioni, procedure o variabili, devono essere identificatori.

La sintassi di una richiesta di funzione è la seguente:

L'ordine dei parametri al momento della chiamata di una funzione corrisponde all'ordine degli argomenti definiti nell'intestazione della funzione.

Le proprietà delle funzioni definite dall'utilizzatore si possono riassumere nelle regole seguenti:

1. Una funzione fornisce *un solo risultato* scalare, sottocampo o puntatore, che dev'essere assegnato all'identificatore della funzione.
2. Una funzione dev'essere dichiarata con un identificatore di un tipo corrispondente a quello del risultato.
3. I parametri, o argomenti, della funzione possono appartenere ad *un tipo qualunque*. Sono detti muti, o formali.
4. Gli oggetti definiti nel corpo della funzione sono locali rispetto a tale funzione. Bisogna evitare di modificare le variabili globali non definite come parametri.
5. La richiesta di una funzione deve effettuarsi per mezzo di *parametri effettivi*; corrispondenti ai parametri formali.

Fra non molto riparleremo delle funzioni ricorsive, dopo aver introdotto le procedure.

1.2.7 - Applicazioni: funzioni matematiche derivate dalle funzioni standard

Si possono definire le seguenti funzioni:

a) *Funzioni trigonometriche inverse*

arc sen (x) = atan(x/sqrt(– x * x + 1))
arc cos (x) = – atan(x/sqrt(– x * x + 1))+ π/2
arc ctg (x) = – atan(x) + π/2

b) *Funzioni iperboliche*

cosh(x) = (exp(x) + exp(–x))/2
senh(x) = (exp(x) – exp(–x))/2
tgh(x) = – exp(–x)/(exp(x) + exp(–x)) * 2 + 1
ctgh(x) = exp(–x)/(exp(x) – exp(–x)) * 2 + 1

c) *Funzioni iperboliche inverse*

arc senh(x) = ln(x + sqrt(x * x + 1))
arc cosh(x) = ln(x + sqrt(x * x – 1))
arc tgh(x) = ln((1 + x)/(1 – x))/2
arc ctg(x) = ln((x + 1)/(x – 1))/2

Come esercizio, si suggerisce di scrivere un programma che permetta di calcolare tutte queste funzioni non standard: per verificare il programma, si prenda spunto dal programma, presentato nel paragrafo 1.1.1.5, che dava il risultato delle funzioni esponenziale e logaritmo a base qualunque.

2 - LE PROCEDURE

Si è visto che le funzioni sono limitate al calcolo di un solo risultato. Ma molto spesso un'elaborazione o un algoritmo particolare dà più di un risultato: ad esempio, le radici di un'equazione di secondo grado. A volte poi si lavora su dati strutturati: ad esempio, nella moltiplicazione di due matrici.

Conviene, per più di un motivo, sviluppare siffatte parti di programma come blocchi a parte. Infatti

- una volta scritto e verificato il blocco suddetto, si può riutilizzarlo più volte nello stesso programma o in altri programmi;
- si evita di duplicare le stesse sequenze d'istruzioni in uno stesso programma;
- il programma è scomposto in moduli secondo un procedimento strutturato e di affinamenti successivi. In questo modo il problema di partenza può venir scomposto in compiti più semplici, concretizzantisi in sottoproblemi per i quali si conoscono le modalità di elaborazione.

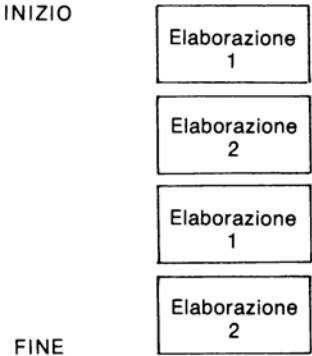
Tutti questi motivi inducono a sviluppare moduli di programma indipendenti, che permettono di soddisfare l'esigenza della strutturazione e scomposizione in blocchi più elementari, e riutilizzabili. Questi moduli sono detti comunemente sottoprogrammi.

In Pascal quest'esigenza è soddisfatta grazie al concetto di *procedura*.

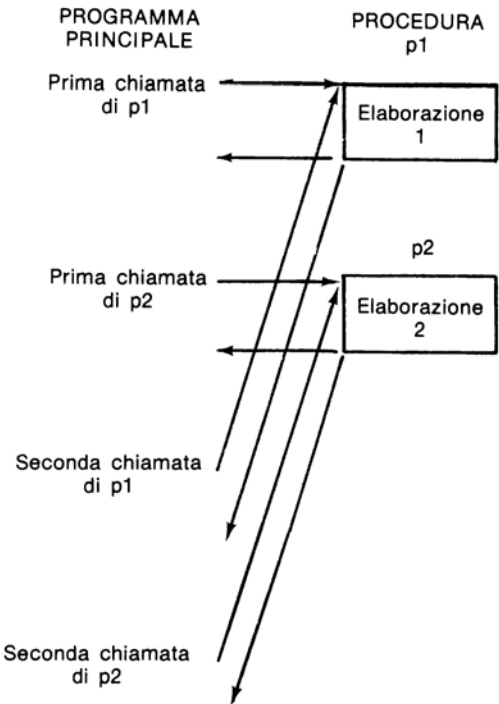
2.1 - Il concetto di procedura

In alcuni programmi accade che la stessa elaborazione venga effettuata più volte in punti diversi. La definizione di funzione permette di richiedere più volte una stessa elaborazione, se questa può essere espressa da un risultato unico; se invece l'elabo-

razione richiede più risultati, le funzioni non si possono usare, e bisogna allora ricorrere ad un sottoprogramma (*subroutine* in inglese), che in Pascal prende il nome di *procedura*.

Supponiamo che un programma abbia questa struttura:

Vediamo che le istruzioni corrispondenti alle elaborazioni 1 e 2 si ripetono.

Quindi è conveniente scrivere una sola volta le istruzioni relative e farvi riferimento durante il programma.

Abbiamo così una struttura del tipo

La ripetizione delle elaborazioni 1 e 2 permette di assimilarle a delle procedure. Allora lo schema precedente rappresenta la nuova struttura, nella quale si ha una sola versione delle istruzioni che compongono le elaborazioni 1 e 2. Questa soluzione presenta due vantaggi:

- permette di evitare la duplicazione delle istruzioni che effettuano un dato trattamento, quando questo è necesario in più punti di uno stesso programma;
- permette di utilizzare parti di programma già verificate che servono per altri programmi, e inoltre di costruire dei nuovi programmi con moduli "prefabbricati".

Una procedura può essere anche vista come una "scatola nera" nella quale vengono introdotti dei dati, e che fornisce dei risultati senza che ci si debba preoccupare di quello che c'è nel blocco della procedura.

2.1.1 - Esempio di procedura

Ci proponiamo di scrivere un programma che scriva per due volte il messaggio "buongiorno".

Invece di ripetere l'istruzione di scrittura del messaggio, definiamo una procedura. Il programma è il seguente:

```
PROGRAM proced;
(*definizione di procedura*)
PROCEDURE buongiorno;
BEGIN
  writeln ('buongiorno');
END;
(*programma principale*)
BEGIN
  buongiorno;
  buongiorno
END.
```

Questo è il più semplice esempio di procedura che si possa dare. In essa non compare nessun parametro. È comunque possibile modificarla introducendovi un parametro che permetterà poi di scrivere i messaggi "buongiorno signora" e "buongiorno signore". Si ha così il programma seguente:

```
PROGRAM proced;
(*definizione di procedura*)
PROCEDURE buongiorno (s: string);
BEGIN
  writeln ('buongiorno', s);
END;
(*programma principale*)
BEGIN
  buongiorno ('signora');
  buongiorno ('signore');
END.
```

Questa procedura può anche essere richiamata con una variabile il cui valore è fornito all'esecuzione del programma tramite un'operazione di lettura, e che permette di scrivere un messaggio tipo "buongiorno x", ove *x* varia ad ogni esecuzione.

Il programma sarà il seguente:

```
PROGRAM proced;
VAR x: string;
(*definizione di procedura*)
PROCEDURE buongiorno (s: string);
BEGIN
  writeln ('buongiorno', s);
END;
(*programma principale*)
BEGIN
  write ('chi è' lei?');
  read (x);
  buongiorno (x)
END.
```

Questi esempi mostrano la semplicità e la potenza insieme del concetto di procedura: semplicità, perchè, per utilizzare una procedura di blocco di programma principale, basta richiederla con il suo nome, seguito o no dai parametri posti fra parentesi; potenza, perchè è chiaro che la complessità del corpo della procedura può essere molto maggiore che negli esempi precedenti.

2.1.2 - Definizione di procedura

Una *procedura* è un blocco di programma che lavora su dati forniti o comunque presenti nel blocco del programma. Questi dati vengono definiti quando si richiede la procedura, e la loro elaborazione permette di avere di ritorno dei risultati ai quali il blocco che ha effettuato la richiesta può accedere. Dati e risultati costituiscono i *parametri*, o *argomenti*, della procedura.

In Pascal tutti i blocchi d'istruzioni possono venir trasformati in procedure mediante una dichiarazione appropriata.

Prendiamo ad esempio la selezione di un elemento di una griglia di selezione del tipo

A: elaborazione a
B: elaborazione b
C: elaborazione c

Abbiamo visto che ciascuna delle possibili scelte può essere considerata come un'istruzione di una struttura CASE; avremo pertanto:

```
PROGRAM alberomenu;
VAR
  selezione: char;
(*radice dell'albero*)
PROCEDURE menu (VAR scelta: char);
BEGIN
  write ('scelta'); read (scelta)
END;
PROCEDURE sceltaa;
  BEGIN
    write ('elaborazione a')
  END;
PROCEDURE sceltab;
  BEGIN
    write ('elaborazione b')
  END;
BEGIN
  menu (selezione);
  CASE selezione OF
    'A': sceltaa;
    'B': sceltab
  END;
END.
```

Il concetto di procedura permette di considerare un gruppo d'istruzioni relativo ad una data elaborazione come un'entità relativamente indipendente, che può venir richiesta dal programma principale.

Contrariamente a quanto avviene per le funzioni, la richiesta di una procedura è in sè e per sè un'istruzione.

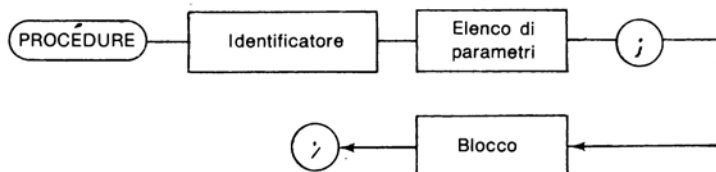
Quindi un programma è scomponibile in blocchi di procedure, anche se queste vi sono utilizzate una sola volta. In tal modo il programma è strutturato meglio, e sarà più leggibile, più comprensibile, e di conseguenza più facilmente modificabile.

2.1.3 - Dichiarazione delle procedure

Così come avviene per le funzioni, ogni procedura dev'essere dichiarata nella sezione appropriata del programma.

Anche una dichiarazione di procedura è costituita da un'*intestazione* e da un *corpo*.

L'intestazione comprende l'identificatore della procedura seguito dall'elenco dei *parametri*: non è necessario definire il tipo della procedura, perchè questa può fornire più risultati di tipi diversi.

Il diagramma sintattico di una dichiarazione di procedura è il seguente:

L'elenco dei parametri è analogo a quello delle funzioni, e può comprendere, fra parentesi,

– identificatori di parametri formali, con il relativo tipo;
– variabili, con il relativo tipo;
– funzioni, con il relativo tipo;
– procedure.

I vari tipi di parametri sono separati da punti e virgola; i parametri invece sono separati da virgole.

Il diagramma sintattico dell'elenco dei parametri è già stato presentato per le funzioni.

Comunque ci soffermiamo ancora sul problema dei parametri, più importante qui che per le funzioni, perchè i risultati della procedura possono venir trasmessi solo in determinati parametri, e non in parametri qualunque; inoltre, i risultati di una procedura possono essere dati strutturati.

I parametri ed i risultati di una procedura si possono trasmettere in due modi:
– ricorrendo a variabili globali;
– trasmettendoli esplicitamente come parametri.

Si tratta di due soluzioni estreme, ma è anche possibile combinare i due metodi.

Nel primo caso la procedura può non avere l'elenco dei parametri.

Nel secondo caso tutti i parametri d'ingresso ed i risultati possono venir trasmessi nell'elenco delle variabili di uscita.

Vediamo a questo proposito dei semplici esami.

2.1.3.1 - Esempio d'uso delle procedure: procedura senza elenco dei parametri

Supponiamo di avere un programma che permetta d'introdurre due parametri quantitativi relativi ad un gruppo d'individui: ad esempio il peso e l'altezza. Si vuole scrivere un programma capace di calcolare la media e la varianza di questi due parametri.

Ricordiamo che, se si hanno n individui, ed x_i esprime la grandezza associata all'individuo i, per definizione la media è

$$m = \frac{\sum_{i=1}^{n} x_i}{n}$$

La varianza teorica è

$$v = \frac{\sum_{i=1}^{n} x_i^2}{n} - m^2$$

e la sua stima è

$$v = \frac{\sum x^2 - (\sum x^2)/n}{n-1}$$

Sappiamo inoltre che lo scarto tipico è

$$s = \sqrt{v}$$

Qui è evidente che il calcolo della media e della varianza dev'essere considerato come una procedura, che verrà richiesta due volte: l'una per il primo parametro, l'altra per il secondo.

Prima di tutto scriviamo la procedura.

Abbiamo già visto un programma di calcolo della media; per calcolare la varianza, basta calcolare la somma dei quadrati Σx_i^2.

La prima versione del programma, senza elenco dei parametri, è la seguente:

```
PROGRAM medvar;
uses transcend;
VAR
   n, i: integer;
   misura: ARRAY[1 ... 100] OF real;
PROCEDURE medivarian;
VAR
   i: integer;
   media, varianza: real;
   s, v: real;
```

```
BEGIN
  s: = 0; v: = 0;
  FOR i: = 1 TO n DO
    BEGIN
      s: = s + misura[i];
      v: = v + sqr(misura[i])
    END;
  media: = s/n;
  varianza: = (v — sqr(s)/n)/(n — 1);
  writeln ('media =',media,'varianza =',varianza);
END;
(*programma principale*)
BEGIN
  write ('numero individui =');
  read (n);
  writeln ('peso');
  FOR i: = 1 TO n DO
    read (misura[i]);
  writeln;
  medivarian;
  writeln ('altezza');
  FOR i: = 1 TO n DO
    read (misura[i]);
  writeln;
  medivarian;
END.
```

La procedura *medivarian* permette di calcolare la media e la varianza con le formule che abbiamo or ora fornito. A carico del programma principale sono la lettura dei dati e la stampa dei risultati. In questa prima versione i dati sono letti in sequenza, riferendosi allo stesso vettore, il cui nome è *misura*: questo vettore è una variabile globale, utilizzata anche nella procedura.

La procedura *medivarian* è richiesta due volte, nel programma principale: la prima volta per il calcolo della media e della varianza dei dati di peso; la seconda volta per i dati di altezza. In tal modo ci siamo risparmiati la fatica di scrivere due volte il calcolo della media e della varianza, ricorrendo ad una procedura.

Un'altra possibilità sarebbe stata scrivere due distinte procedure, denominate *media* e *varianza*, o anche due funzioni, perchè in questo caso si ottiene un unico risultato.

In ogni caso entrambe le soluzioni consentono un *risparmio di tempo* a livello di programmazione ed un *risparmio di spazio in memoria*, perchè evitano la duplicazione delle stesse istruzioni

Nell'esempio riportato abbiamo a che fare sia con delle *variabili globali*, ossia

– il vettore *misura*;
– il numero di individui *n*;
– la variabile di controllo *i*;

sia con delle *variabili locali*, che sono

– le variabili di lavoro *s* e *v*;
– le variabili di risultato *media* e *varianza*;
– la variabile di controllo *i*.

Sottolineiamo che la procedura *medivarian* impiega le variabili globali *misura* ed *n*.

Invece la variabile *i* che viene ridefinita all'interno della procedura è una variabile locale diversa dalla variabile globale *i* del programma principale, anche se nei due casi la funzione è la stessa. È chiaro che, per evitare ambiguità, avremmo potuto dar loro due nomi diversi, ma si tenga presente che, dal momento in cui una variabile viene dichiarata localmente, ad essa si può accedere soltanto nell'ambito della procedura in cui è stata definita. Se però le variabili locale e globale hanno lo stesso nome, e ci si dimentica di dichiarare localmente la variabile, questa sarà considerata come una variabile globale, con un grosso rischio di spiacevoli *effetti margine*, cosi come avviene per le funzioni.

Pertanto è opportuno definire tutte le variabili locali con dichiarazioni appropriate, all'interno della procedura. Così facendo, il modulo del programma sarà indipendente dal contesto del programma richiedente.

Questo primo esempio rappresenta certamente un passo avanti rispetto alla duplicazione delle istruzioni che permettono di calcolare la media e la varianza; si tenga presente tuttavia che questa procedura non è completamente indipendente dal programma che la richiede, perchè utilizza le variabili globali *n* e *misura*.

2.1.3.2 - Procedura con elenco dei parametri

Per rendere la procedura dell'esempio precedente più generale, occorre modificarla introducendo come parametro il numero di elementi ed il vettore dei dati sui quali la procedura medesima lavora. Avremo così il programma

```
PROGRAM medvar;
uses transcend;
TYPE
   elenco = ARRAY[1 ... 100] OF real;
VAR
   n, i: integer;
   misura: elenco;
   m, v: real;
PROCEDURE medivarian (VAR media, varianza: real;
   vet: elenco; n: integer);
```

```
VAR
  i: integer;
  s, v: real;
BEGIN
  s: = 0; v: = 0;
  FOR i: = 1 TO n DO
    BEGIN
      s: = s + vet[i];
      v: = v + sqr(vet[i])
    END;
  media: = s/n;
  varianza: = (v - sqr(s)/n)/(n - 1);
END;
  (*programma principale*)
BEGIN
  write ('numero individui =');
  read (n);
  writeln ('peso');
  FOR i: = 1 TO n DO
    read (misura[i]);
  writeln;
  medivarian (m, v, misura, n);
  writeln ('media =', m, 'varianza =', v);
  writeln;
  writeln ('altezza');
  FOR i: = 1 TO n DO
    read (misura [i]);
  writeln;
  medivarian (m, v, misura, n);
  writeln ('media =', m, 'varianza =', v);
END.
```

Esecuzione

```
numero individui = 5
peso
60
55
45
65
50
media = 5.50000E1 varianza = 6.25000E1
altezza
162
```

```
150
145
160
155
media = 1.54400E2 varianza = 4.93008E1
```

Si osservi che in questo programma le modifiche sono minime, in quanto si è soltanto aggiunto un elenco di parametri formali.

All'interno della procedura, il parametro formale relativo al numero di elementi si chiama *n*, il vettore dei dati di lavoro *vet*. Infatti non c'è più alcun motivo per mantenere la denominazione *misura*, perchè siamo di fronte ad una procedura generale, che permette di calcolare la media e la varianza di un qualunque vettore di dati.

Peraltro i risultati del calcolo eseguito nella procedura non sono più scritti all'interno di quest'ultima, ma vengono anch'essi trasmessi come parametri risultato, sotto forma di *variabili* denominate *media* e *varianza*.

Invece le variabili *s* e *v* restano locali rispetto alla procedura, e quindi non saranno note all'esterno di essa. Perchè siano accessibili al programma o alla procedura richiamante, i parametri risultato devono essere definiti come *parametri variabili*.

2.1.4 - Norme sulla trasmissione dei parametri di una procedura

Sono regole analoghe a quelle relative alle funzioni, tranne che per il fatto che qui si hanno parametri risultato.

Ricordiamo quindi le regole relative alla trasmissione dei parametri di una procedura.

a) Nella dichiarazione, l'elenco dei parametri è un elenco di parametri formali (muti), il cui tipo dev'essere definito da un identificatore di tipo.

b) Al momento della chiamata di una procedura, che avviene richiamando il nome della procedura stessa, l'elenco dei parametri è sostituito da un ugual numero di parametri effettivi, corrispondenti in modo biunivoco all'ordine ed al tipo dei parametri formali.

c) Esistono quattro tipi di parametri argomento:
   - parametri trasmessi mediante il loro *valore*: allora i parametri effettivi trasmessi possono essere delle *espressioni* (valore dell'espressione);
   - parametri *variabili*, trasmessi sotto forma di identificatori di variabili, ed il cui contenuto potrà venir modificato dalla procedura. I parametri *risultato* devono essere definiti formalmente come parametri variabili;
   - parametri *funzione*, rappresentati di fatto da un identificatore di funzione, standard o non standard;
   - parametri *procedura*, rappresentati di fatto da un identificatore di procedura, standard o non standard.

d) Il calcolo degl'indirizzi avviene al momento della richiesta della procedura; in particolare, per gli elementi di vettori, è a questo punto che viene calcolato l'indice. Ad ogni chiamata di procedura, gl'indirizzi degli effettivi parametri variabili diventano quelli dei parametri formali.

Invece i parametri definiti dai loro *valori* corrispondono a variabili *locali*, inizializzate con i valori, al momento della chiamata della procedura, del parametro effettivo (o con il valore dell'espressione associata). Anche nel caso di un identificatore di variabile il contenuto di questa *non sarà modificato* al rientro dalla procedura: quindi questo tipo di parametro non si può usare per la trasmissione dei risultati.

Esempio

```
PROGRAM valvar;
VAR
  c: char;
PROCEDURE valore (car: char);
BEGIN
  car: = succ (car);
  writeln (car)
END;
PROCEDURE variabile (var car: char);
BEGIN
  car: = succ (car);
  write (car)
END;
BEGIN
  read (c);
  valore (c); write (c);
  variabile (c); write (c)
END.
```

Se come dato è letto il carattere *k*, si ha:

```
l    k
l    l
```

Nella procedura *valore* la variabile non è stata modificata; nella procedura *variabile* la modifica è trasmessa al programma che l'ha richiamata.

OSSERVAZIONI

a) Quando un parametro viene trasmesso tramite il suo valore, quest'ultimo è copiato in una variabile locale. Questo avviene anche nella trasmissione per valo-

re di un vettore: ne consegue un tempo di copia ad ogni chiamata di procedura. Quindi, nella trasmissione di vettori, per risparmiar tempo è meglio optare per la trasmissione per parametri variabili.

b) Al momento della chiamata di una procedura, il contenuto delle variabili locali è indeterminato.

c) Bisogna stare attenti alle modifiche operate sulle variabili globali da una procedura. In genere è da preferirsi la trasmissione per parametri variabili.

2.1.5 - Il corpo di una procedura

Il corpo di una procedura è un blocco, e di conseguenza può contenere tutte le dichiarazioni e le istruzioni del linguaggio.

Nel corpo di una procedura tutti gli oggetti non trasmessi come parametri, e che non siano neppure delle variabili globali, devono essere dichiarati.

Dal momento che la definizione di un blocco è ricorsiva, è possibile definire delle procedure (o delle funzioni) all'interno di una procedura (o di una funzione).

Sono possibili annidamenti a più livelli di funzioni e di procedure. In tal caso, al fine di orientarsi e di documentare il programma, è opportuno indicare per mezzo di un commento l'inizio e la fine di ciascun blocco.

Esempio

```
PROGRAM procedurannidate;
VAR
  n0: type 0;
PROCEDURE livello 1;
VAR
  n1: type 1;
PROCEDURE livello 2;
VAR
  n2: type 2;
BEGIN (*corpo procedura livello 2*)
  istruzioni 2
END; (*fine procedura livello 2*)
BEGIN (*corpo procedura livello 1*)
  istruzioni 1;
END; (*fine procedura livello 1*)
BEGIN (*programma livello 0*)
  istruzioni 0
END. (*fine programma*)
```

2.1.6 - Chiamata di procedure

Si è visto che la *chiamata* di una procedura costituisce un'istruzione formata dal nome della procedura, seguito eventualmente dalla sequenza dei parametri effettivi, fra parentesi, e separati da virgole.

In alcuni casi è possibile richiamare una procedura non ancora definita, segnalandolo però con la dichiarazione *forward* (*oltre*), che indica che la procedura è definita in un punto successivo del programma.

Questo vale anche per le funzioni quando vengono richieste prima di essere state definite.

Esempio

```
PROGRAM oltre;
VAR
  a, b: char;
PROCEDURE dopo (c: char); forward;
PROCEDURE prima (car: char);
BEGIN
  write (car);
  dopo (car)
END;
PROCEDURE dopo;
BEGIN
  writeln (c);
  write ('dopo =');
  readln (c);
  IF c < > '#' THEN prima (c);
END;
BEGIN
  readln (a, b);
  WHILE NOT eof DO
    BEGIN
      prima (a);
      dopo (b);
      readln (a, b);
    END;
END.
```

La procedura *dopo*, con il relativo elenco di parametri, è annunciata da una dichiarazione *forward*; successivamente viene definito il blocco d'istruzioni della procedura *prima*, che fa riferimento alla procedura *dopo*. Infine viene il corpo della procedura *dopo*: in questo caso è inutile ridefinire la lista dei parametri della procedura *dopo*.

Nell'esempio in questione anche la procedura *dopo* fa riferimento a sua volta alla

procedura *prima*. Se così non fosse stato, la dichiarazione *forward* non sarebbe servita, in quanto sarebbe stato sufficiente invertire l'ordine delle procedure *prima* e *dopo*.

In conclusione, è utile usare la dichiarazione *forward* quando due procedure si richiamano reciprocamente: questo fenomeno prende anche il nome di ricorsività incrociata.

L'esecuzione del programma che precede darà ad esempio

```
XY Ⓡ
XX
dopo = A Ⓡ
AA
dopo = B Ⓡ
BB
dopo = C Ⓡ
CC
dopo = # Ⓡ
Y
dopo = X Ⓡ
XX
dopo = # Ⓡ
AB Ⓡ
AA
dopo = D Ⓡ
DD
dopo = # Ⓡ
B
dopo = # Ⓡ
```

2.2 - Applicazione: un programma di conversione

Nel programma di conversione (da decimale a binario) disponiamo i valori entro un vettore.

Il programma è il seguente, in cui è definita anche una procedura di elevamento a potenza:

```
PROGRAM conversionebinaria;
TYPE
   binario = 0 ... 1;
   bit = ARRAY[0 ... 15] OF binario;
VAR
   i, k, n: integer;
   b: bit;
FUNCTION pot(b, j: integer): integer;
```

```
VAR
  p, i: integer;
BEGIN
  IF j = 0 THEN p: = 1
    ELSE
    BEGIN
      p: = b;
      FOR i: = 2 TO j DO p: = p * b;
    END;
  pot: = p;
END;
PROCEDURE bin (var b: bit; n, k: integer);
VAR
  i: integer;
BEGIN
  FOR i: = k - 1 DOWNTO 0 DO
    BEGIN
      IF n < pot(2, i) THEN b[i]: = 0
        ELSE b[i]: = 1;
      n: = n - b[i] * pot(2, i);
    END;
END;
BEGIN
  readln (n);
  k: = 15;
  bin (b, n, k);
  FOR i: = k - 1 DOWNTO 0 DO write (b[i]);
END.
```

3 - LA RICORSIVITÀ DELLE PROCEDURE E DELLE FUNZIONI

In un capitolo precedente abbiamo introdotto il concetto di ricorsività, ma non l'abbiamo ancora utilizzato.

Nella definizione di una procedura o di una funzione ci si può riferire, nel blocco delle istruzioni, a questa stessa procedura (o funzione). Si ha allora quella che vien detta una procedura (o funzione) ricorsiva.

3.1 - Definizione

L'uso del nome (identificatore) di una procedura (o di una funzione) nel corpo di questa procedura (o funzione) comporta un'esecuzione ricorsiva della procedura (o funzione) medesima.

In Pascal possiamo scrivere delle procedure ricorsive perchè ad ogni richiesta della procedura vengono generate delle nuove variabili locali.

Infatti la ricorsività comporta il salvataggio dei risultati intermedi: se le variabili locali fossero le stesse ad ogni richiesta, la ricorsività non sarebbe possibile.

3.2 - Le funzioni ricorsive

Diamo innanzitutto qualche esempio di funzioni ricorsive, cominciando da quelle che possono essere scritte nella forma

$$f_n(x) = g(f_{n-1}(x)) \qquad \text{per } n > 0$$

1) La funzione elevamento a potenza, espressa da

$$x^n = x \cdot x^{n-1} \qquad \text{con } x^0 = 1$$

può essere definita ricorsivamente in questo modo:

```
FUNCTION potenza(n: integer; x: real):  real;
BEGIN
  IF n = 0 THEN potenza: =1'
    ELSE potenza: = potenza(n - 1, x)* x
END;
```

2) Analogamente, per la funzione fattoriale, espressa da

$$n! = n \times (n - 1)! \qquad \text{con } 0! = 1$$

avremo:

```
PROGRAM fattric;
VAR n: integer;
(*definizione ricorsiva di fattoriale*)
FUNCTION fatt(n: integer): integer);
BEGIN
  IF n = 0 THEN fatt: = 1 ELSE fatt: = n * fatt(n - 1);
END;
BEGIN
  writeln ('introdurre un numero');
  read (n);
  writeln ('fattoriale',n,'=',fatt(n) );
END.
```

3) Un esempio più complesso è rappresentato dalla *funzione di Ackermann*, definita come segue:

A(m,n) = A(m − 1,A(m,n − 1)) per m ed $n > 0$
A(0,n) = n + 1
A(m,0) = A(m − 1, 1) per $m > 0$

Questo è il programma:

```
PROGRAM Ackermann;
VAR
  n, m: integer;
FUNCTION Acker(m, n: integer): integer;
BEGIN
  IF m = 0 THEN Acker: = n + 1
    ELSE IF n = 0 THEN Acker: = Acker(m − 1, 1)
                ELSE Acker: = Acker(m − 1, Acker(m, n − 1))
END;
  (*programma principale*)
BEGIN
  FOR m: = 0 TO 3 DO
    FOR n: = 0 TO 3 DO
      writeln ('Ackermann',m,' ',n,=',Acker(m, n));
END.
```

Esecuzione

```
Ackermann 0   0 =  1
          0   1 =  2
          0   2 =  2
          0   4 =  5
          1   0 =  2
          1   1 =  4
          1   3 =  5
          1   4 =  6
          2   0 =  3
          2   1 =  5
          2   2 =  7
          2   3 =  9
          2   4 = 11
          3   0 =  5
          3   1 = 13
          3   2 = 29
          3   3 = 61
```

Questa funzione diventa ben presto difficilmente calcolabile in maniera ricorsiva, perchè il sistema di impilamento delle chiamate ricorsive rischia di saturarsi rapidamente. Per questa ragione questa funzione è un buon test del livello di ricorsività possibile su un sistema.

4) Il M.C.D. è definibile ricorsivamente, in virtù della proprietà

MCD(a, b) = MCD(b, a mod b)

Pertanto avremo:

```
PROGRAM mcdricorsivo;
VAR a, b: integer;
FUNCTION mcd(i, j: integer): integer;
BEGIN
  IF j = 0 THEN mcd: = i
    ELSE mcd: = mcd(j,i MOD j)
END;
BEGIN
  writeln ('introdurre due numeri');
  read (a, b);
  write ('mcd di',a,'e',b,'=',mcd(a, b) );
END.
```

Vediamo che la funzione MCD soddisfa esattamente la proprietà or ora enunciata, e, inoltre, che una funzione può essere richiesta nell'argomento di un'istruzione di scrittura.

Il M.C.D. può essere ancora definito per mezzo della proprietà

MCD(a, b) = MCD(b, a − b) se $a > b$
MCD(a, b) = mcd(a, b − a) se $a < b$

Avremo allora il programma seguente:

```
PROGRAM mcd2;
VAR a, b: integer;
FUNCTION mcd(i, j: integer): integer;
BEGIN
  IF i = j THEN mcd: = i ELSE
  BEGIN
    IF i < j THEN mcd: = mcd(j - i, i)
            ELSE mcd: = mcd(i - j, j);
  END;
END;
BEGIN
  writeln ('introdurre due numeri');
  read (a, b);
  (*calcolo del mcd*)
  write ('mcd di',a,'e',b,'=',mcd(a, b) );
END.
```

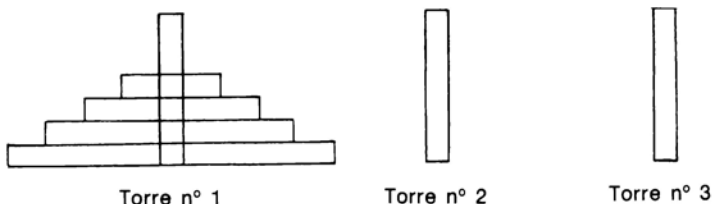
3.3 - Le procedure ricorsive

Finora non ci siamo ancora imbattuti in problemi tipicamente ricorsivi. Orbene, per convincerci che alcuni problemi si risolvono più facilmente in questo modo, esaminiamo adesso un problema classico di programmazione ricorsiva.

3.3.1 - Le torri di Hanoi

Si tratta di un gioco, e una leggenda, dell'estremo oriente. Dei monaci buddisti giocano con dei paletti e dei dischi, e la fine della partita è vista come il simbolo della fine del mondo.

I giocatori hanno a disposizione tre paletti, o torri, nei quali possono infilarsi dei dischi, sovrapposti in modo che i loro diametri siano via via più piccoli. I cerchi sono 64.

La situazione di partenza si può raffigurare così:

Il gioco consiste nel trasferire i dischi dalla torre 1 alla torre 3, spostandone uno solo per volta ed utilizzando la torre di mezzo, la 2, affinchè in nessun momento un disco sia posto su uno più piccolo.

Il gioco è molto facile da capire, e il nostro problema è trovare un algoritmo capace di riprodurlo nel rispetto delle regole.

È evidente che l'operazione di base del gioco consiste nello spostare un disco da una torre ad un'altra, per cui si può pensare ad una procedura a due parametri: il primo è la torre di partenza, il secondo la torre di arrivo.

Questa procedura potrà ad esempio essere denominata

disco (torrepartenza, torrearrivo)

Il suo effetto sarà quello di spostare un disco da una torre ad un'altra.

Per risolvere il problema proposto con *n* dischi, possiamo ad esempio supporre che $n - 1$ dischi siano stati trasferiti alla torre 2, passando per la torre 3, e che quindi il disco più grande si trovi nella torre 1.

Possiamo quindi trasferire questo disco dalla torre 1 alla torre 3 mediante la procedura *disco*, ed infine trasferire gli $n - 1$ dischi dalla torre 2 alla torre 3 utilizzando la torre 1, che in questa fase è vuota, come torre di deposito temporaneo.

A questo punto il problema consiste nel realizzare una procedura capace di trasferire una pila di $n - 1$ dischi da una torre di partenza ad una torre di arrivo passando per una torre intermedia.

Quest'operazione si può rappresentare con la procedura

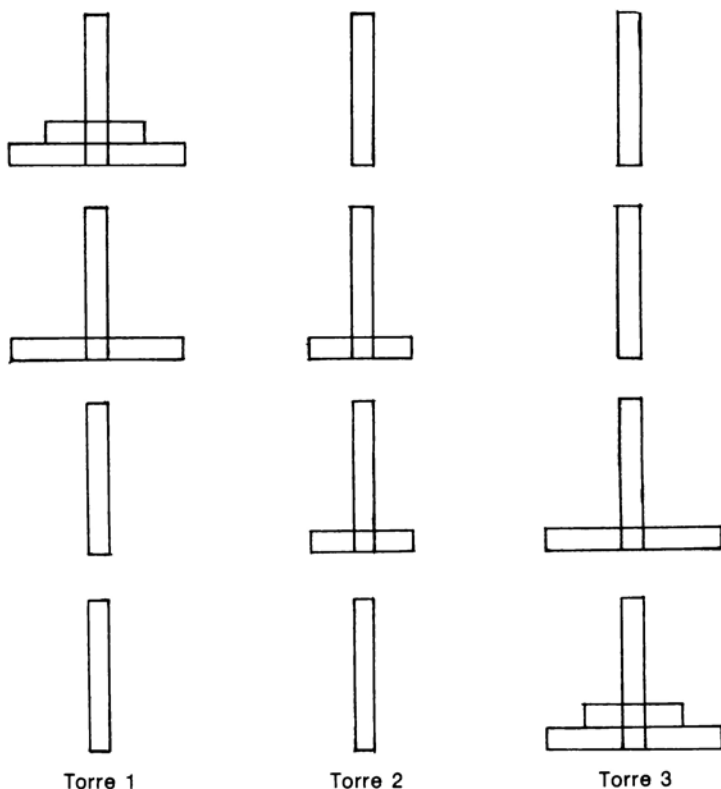
torre (k, torrepartenza, torrearr, torreint)

ove *k*, è il numero dei dischi, e gli altri parametri sono i numeri d'ordine delle torri.

Il problema iniziale era trasferire *n* dischi dalla torre 1 alla torre 3 passando per la torre 2.

Il problema è scomponibile in tre richieste di procedura:

torre (n – 1, torre 1, torre 2, torre 3)
disco (torre 1, torre 3)
torre (n – 1, torre 2, torre 3, torre 1)

Il problema è allora risolto, a patto di conoscere il modo di procedere con una torre formata da uno o due dischi: in effetti, definendo una procedura *torre* in forma ricorsiva, siamo ricondotti al problema dello spostamento di una pila di due dischi da una torre ad un'altra passando per una terza torre. E questo sappiamo come farlo, come si vede nello schema seguente:

Ed ecco la prima versione del programma:

```
PROGRAM torrehanoi;
VAR
  ndisco: integer;
PROCEDURE torre (nrdis, dallatorre, allatorre,
  perlatorre: integer);
  (*spostamento di un disco*)
  PROCEDURE disco (torrepart, torrearr: integer);
  BEGIN
    writeln ('spostare il disco da', torrepart,'a',torrearr)
  END;
(*corpo della procedura torre*)
BEGIN
  IF nrdis > 0 THEN
  BEGIN
    torre (nrdis – 1, dallatorre, perlatorre, allatorre);
    disco (dallatorre, allatorre);
    torre (nrdis – 1, perlatorre, allatorre, dallatorre)
  END;
END
(*programma principale*)
BEGIN
  write ('introdurre il numero dei dischi');
  read (ndisco);
  torre (ndisco, 1, 3, 2);
END.
```

Supponendo di avere un solo disco, e quindi di volerlo spostare da una torre ad un'altra, in tal caso la procedura *torre* si riduce come caso particolare alla procedura *disco*, che si limita a scrivere, ad ogni mossa, lo spostamento effettuato.

I parametri della procedura *torre* sono: il numero dei dischi costituenti la torre, il nome della torre di partenza (qui, simbolicamente, *dallatorre*), il nome simbolico della torre di arrivo (*allatorre*), ed il nome della torre intermedia (*perlatorre*).

Avremo allora i seguenti risultati:

```
1.    introdurre il numero dei dischi 3 Ⓡ
      spostare il disco da 1 a 3
      spostare il disco da 1 a 2
      spostare il disco da 3 a 2
      spostare il disco da 1 a 3
      spostare il disco da 2 a 1
      spostare il disco da 2 a 3
      spostare il disco da 1 a 3
```

2. introdurre il numero dei dischi 4 Ⓡ
spostare il disco da 1 a 2
spostare il disco da 1 a 3
spostare il disco da 2 a 3
spostare il disco da 1 a 2
spostare il disco da 3 a 1
spostare il disco da 3 a 2
spostare il disco da 1 a 2
spostare il disco da 1 a 3
spostare il disco da 2 a 3
spostare il disco da 2 a 1
spostare il disco da 3 a 1
spostare il disco da 2 a 3
spostare il disco da 1 a 2
spostare il disco da 1 a 3
spostare il disco da 2 a 3

Questo programma ha il pregio di descrivere l'algoritmo come è stato concepito mediante il ragionamento semplice che abbiamo descritto precedentemente.

Il programma può essere semplificato tenendo conto del fatto che lo spostamento di un disco è equivalente a quello di una torre con un disco, e che quindi la procedura *disco* può essere considerata come un caso particolare della procedura di spostamento di una torre. È chiaro che allora la torre intermedia diventa inutile, e che i primi due parametri rappresentano le torri di partenza e di arrivo.

Quindi la chiamata della procedura *disco* è equivalente a quella della procedura seguente:

```
torre (1, torrepartenza, torrearrivo, torreint)
```

Possiamo così riscrivere il programma in una forma più concisa:

```
PROGRAM torrehanoi
VAR
  ndisco: integer;
PROCEDURE torre (nrdis, dallatorre, allatorre,
  perlatorre: integer);
(*corpo della procedura torre*)
BEGIN
  IF nrdis = 1 THEN
    writeln ('spostare il disco da',dallatorre,'a',allatorre)
  ELSE
  BEGIN
    torre (nrdis – 1, dallatorre, perlatorre, allatorre);
```

```
      torre (1, dallatorre, allatorre, perlatorre);
      torre (nrdis — 1, perlatorre, allatorre, dallatorre)
    END;
  END;
  (*programma principale*)
  BEGIN
    write ('introdurre il numero dei dischi');
    read (ndisco);
    torre (ndisco, 1, 3, 2);
  END.
```

La lettura di questo programma, immediatamente comprensibile ricordando che è ricorsivo, può dare l'impressione che il problema non sia veramente risolto. Per convincersi del contrario, basta simularlo a tavolino! Il lettore avrà modo di apprezzare l'utilità della ricorsività cercando di scrivere un programma non ricorsivo capace di risolvere lo stesso problema.

La ricorsività trova esempi tipici di applicazione solo in alcuni giochi? No di certo. Lo vedremo esaminando i due esempi seguenti, dei quali l'uno è un calcolo matematico classico, cioè il calcolo del determinante di una matrice, l'altro è un tipico problema informatico concernente un algoritmo di ordinamento rapido.

3.3.2 - Calcolo del determinante di una matrice quadrata

Un sistema di equazioni lineari è rappresentabile con due membri: nel primo compaiono un vettore $X = (x1, x2 \ldots xn)$ ed una matrice A, che rappresenta i coefficienti di ciascuna equazione; il secondo è costituito dal vettore $B = (b1, b2 \ldots bn)$.

Pertanto il sistema di equazioni

$$
\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\
&a_{21}x_1 + a_{22}x_2 + \ldots + a_{1n}x_n = b_2 \\
&\qquad\ldots\ldots\ldots\ldots \\
&a_{n1}x_1 + \ldots \qquad\qquad + a_{nn}x_n = b_n
\end{aligned}
$$

$$
\text{se } A = \begin{pmatrix} a_{11} \ldots\ldots a_{1n} \\ a_{21} \ldots\ldots a_{2n} \\ \ldots\ldots \\ a_{1n} \ldots\ldots a_{nn} \end{pmatrix}
$$

è rappresentato in notazione matriciale con

$$A \cdot X = B$$

La soluzione del sistema, se non è degenere, è data da

$$X = A^{-1}B$$

Per sapere se esiste una soluzione non degenere, bisogna calcolare il determinante della matrice *A*: se il determinante è diverso da zero, sappiamo che tale soluzione esiste, cioè, in altre parole, che esiste la matrice inversa di *A*, che è A^{-1}.

Il calcolo del determinante è un'operazione definita dall'insieme delle permutazioni dei coefficienti della matrice.

Prima di tutto occorre definire quello che chiamiamo *cofattore*: un cofattore con indici *i*, *j* è definito come l'insieme dei coefficienti della matrice nella quale la riga *i* e la colonna *j* sono state soppresse.

Quindi possiamo definire il calcolo di un determinante con l'espressione

$$\text{determinante}(A) = \sum_{j=1}^{n} (-1)^{i+j} A_{ij} \text{ determinante } (\text{cofattore}_{ij}(A))$$

Questa è una definizione tipicamente ricorsiva, perchè il calcolo di un determinante di ordine *n* comporta il calcolo di *n* determinanti di ordine $n - 1$.

In particolare, è possibile sviluppare un determinante sulla base di una riga qualunque. Se la riga è l'ultima, avremo:

$$\text{determinante}(A) = \sum_{j=1}^{n} (-1)^{n+j} \times A_{nj} \text{ determinante } (\text{cofattore}_{nj}(A))$$

Nel programma che segue viene definita una funzione ricorsiva *deter*, che contiene la procedura *cofattore*, capace di calcolare i cofattori.

Il determinante è ottenuto sulla base dell'ultima riga.

Si osservi la trasmissione dei parametri della funzione *deter* e della procedura *cofattore*.

```
PROGRAM determinante;
TYPE
   matrice = ARRAY[1 ... 20, 1 ...20] OF real;
VAR
   mat: matrice;
   n, r, c: integer;
FUNCTION deter (mat: matrice; n: integer): real;
```

```
VAR
   cof: matrice;
   det: real;
   i, j, segno: integer;
PROCEDURE cofattore (VAR cof: matrice; mat: matrice;
   n, c: integer);
VAR
i, j: integer;
BEGIN
   FOR i: = 1 TO n - 1 DO
   BEGIN
      FOR j: = 1 TO c - 1 DO
         cof[i, j]: = mat[i, j];
      FOR j: = c TO n - 1 DO
         cof [i, j]: = mat[i, j+1];
   END;
END;
BEGIN
   IF n = 1 THEN det: = mat[1, 1]
      ELSE
      BEGIN
         det: = 0; segno: = 1;
         FOR j: = n DOWNTO 1 DO
            BEGIN
               cofattore (cof, mat, n, j);
               det: = det + segno * mat[n, j] * deter (cof, n - 1);
               segno: = - segno;
            END;
      END;
   deter: = det
END;
BEGIN
   read (n);
   FOR r: = 1 TO n DO
      BEGIN
         FOR c: = 1 TO n DO
            read (mat[r, c]);
      writeln;
      END;
      writeln ('determinante =', deter (mat, n));
END.
```

È possibile estrarre la procedura *cofattore* dalla funzione *deter*?

3.3.3 - Algoritmo di ordinamento per partizione (ordinamento rapido)

L'assunto di base di quest'algoritmo è quello di dividere in due parti l'insieme dei valori da ordinare. Nella prima parte compaiono i valori maggiori dell'elemento con cui si opera il confronto, nella seconda compaiono i valori minimi del suddetto elemento.

Ciascuna parte può essere a sua volta considerata come una sequenza di valori da ordinare.

L'algoritmo sarà definito come procedura ricorsiva: il procedimento si arresta quando in una parte rimangono due soli elementi, che a questo punto si possono ordinare, ove sia necessario, con una semplice permutazione.

L'algoritmo richiede che si parta da un valore iniziale che permetta di effettuare la bipartizione: chiameremo le due parti rispettivamente alta e bassa.

La situazione ottimale è che il primo valore sia vicino alla metà della sequenza dei valori da ordinare; in tal modo si hanno due parti di dimensioni simili.

Se la sequenza di valori da ordinare è casuale, sarà opportuno scegliere il valore mediano della sequenza di numeri. Allora, se b ed a sono i limiti di una parte, sceglieremo come valore di confronto il valore

$$m = \text{parte intera} \left(\frac{b + a}{2}\right)$$

Il principio dell'algoritmo consiste nello spostarsi verso l'alto della partizione finquando si trovano valori inferiori al valore di paragone, e verso il basso finquando i valori incontrati sono superiori al valore mediano. Si continua così fino a trovare $b > a$.

Poi si ordinano le due partizioni così ottenute, richiamando la stessa procedura di ordinamento usata per determinare le due partizioni (*basso*, *a*) e (*b*, *alto*).

Il programma è il seguente:

```
PROGRAM ordinarapido;
VAR
  n, i: integer;
  c: ARRAY[1 ... 100] OF integer;
PROCEDURE rapidordina (basso, alto: integer);
VAR
  b, a, m, t: integer;
BEGIN
  b: = basso;
  a: = alto;
  m: = (b + a) DIV 2;
  REPEAT
    WHILE c[m] < c[a] DO a: = a – 1;
    WHILE c[b] < c[m] DO b: = b + 1;
    IF b <= a THEN
```

```
            BEGIN
               t: = c[b];
               c[b]: = c[a];
               c[a]: = t;
               a: = a -1;
               b: = b + 1;
            END;
      UNTIL b > a;
      (*esaminare la parte bassa*)
      IF basso < a THEN rapidordina (basso, a);
      (*esaminare la parte alta*)
      IF b < alto THEN rapidordina (b, alto);
   END;
   (*programma principale*)
   BEGIN
      read (n);
      FOR i: = 1 TO n DO read (c[i]);
      rapidordina (1, n);
      writeln;
      FOR i: = TO n DO write (c[i], ' ')
   END.
```

4 - UN ESERCIZIO DI STILE: IL PROBLEMA DELLE OTTO REGINE

In chiusura del capitolo, esaminiamo un problema un po' più complesso. Nel gioco degli scacchi è possibile sistemare sulla scacchiera otto regine in modo tale che nessuna sia minacciata da un'altra. Ricordiamo che negli scacchi la regina controlla tutte le posizioni situate sulla stessa riga, sulla stessa colonna e sulle due diagonali che passano per la posizione da essa occupata. Il problema prevede più soluzioni, ed è stato per lungo tempo oggetto di ricerche, sia da parte dei giocatori di scacchi che da parte dei matematici, tanto che a metà 800 erano già registrate quaranta soluzioni.

Il matematico Gauss riteneva che le soluzioni fossero settantasei. Oggi sappiamo che ne esistono novantadue, ma a questa conclusione si è giunti per semplice enumerazione: non ci risulta che ne esista una dimostrazione matematica. Ora, ci proponiamo di scrivere un programma in Pascal che permetta di scoprire queste soluzioni.

Consigliamo al lettore di formulare innanzitutto un algoritmo che porti a questo risultato. Anche se il programma non è molto lungo, pensiamo comunque che si tratti di un problema abbastanza complesso, e che quindi sia necessario darne un'analisi piuttosto dettagliata.

4.1 - Definizione di posizione inattaccabile

È chiaro che, anche con un calcolatore, non sarebbe sensato verificare tutte le

possibili configurazioni della distribuzione di otto regine sulla scacchiera: si pensi che, già con una sola regina per colonna, le possibilità per le quali bisognerebbe cercare le soluzioni opportune sono più di 10^9!

Di conseguenza occorre individuare una strategia di collocazione delle regine che tenga conto delle regine già collocate. È ovvio che non si può porre una regina su una colonna o su una riga su cui si trova già un'altra regina, quindi l'algoritmo in questione deve avere come presupposto l'impossibilità di collocare più di una regina per colonna. Come secondo passo, l'algoritmo dovrà cercare, entro ciascuna colonna, la riga o le righe che possono andar bene. Infine, prima di collocare una regina su una data riga, bisognerà accertarsi che non sia minacciata su una diagonale!

Una posizione che risponde a tutti questi requisiti vien detta *posizione inattaccabile*: è una posizione che permette di collocare una nuova regina senza che questa sia minacciata dalle regine già sistemate.

4.1.1 - I principi dell'algoritmo

Se arriviamo all'ottava colonna e troviamo una posizione inattaccabile, abbiamo ottenuto una soluzione che obbedisce ai termini del problema.

Ma bisogna trovarle tutte. A tal fine utilizzeremo un algoritmo detto di *percorso a ritroso* (backtracking in inglese), secondo il quale quando, in un qualunque passaggio dell'algoritmo stesso, per una data colonna si trova (o non si trova) una soluzione, si retrocede alla colonna precedente, per cercare di trovare in questa un'altra posizione inattaccabile; poi si riparte, avanzando fino all'ultima colonna o fino ad una situazione d'impossibilità. Quando l'operazione si è ripetuta fino alla colonna 1, vuol dire che in questa colonna non ci sono altre soluzioni. Allora bisogna cambiare il posto occupato dalla regina nella colonna 1 e ripetere il procedimento finchè la regina che occupa la colonna 1 non sia stata collocata su tutte le otto righe corrispondenti.

Possiamo quindi riassumere l'algoritmo nelle seguenti fasi:

1. Porre la prima regina sulla colonna 1 ($c = 1$).
2. Se $c \geq 8$, stampare la soluzione, e saltare alla fase 5.
3. Porre una regina sulla riga 1 della colonna c ($r = 1$).
4. Se una posizione è inattaccabile, passare alla colonna seguente ($c = c + 1$), e saltare alla fase 2.
5. Se la regina della colonna c è sulla riga 8 ($r = 8$), passare alla colonna precedente ($c = c - 1$).
6. Se $c = 1$ ed $r = 8$: fine.
   Sennò fare $r = r + 1$.
7. Se $r > 8$ saltare alla fase 5.
   Sennò saltare alla fase 4.

Quest'algoritmo è conciso ma non strutturato, per cui non bisogna avventurarsi nella programmazione senza riflettere!

4.2 - La programmazione

La programmazione si differenzia dall'algoritmo or ora illustrato per il fatto che abbiamo utilizzato un vettore *col* per rappresentare la posizione di una regina in una colonna. Quindi *col* [*i*] rappresenta l'*i-esima* colonna, e il suo valore indica la riga di questa colonna sulla quale si trova la regina: ad esempio, se *col* [*i*] = *3*, la regina dell'*i-esima* colonna si trova sulla terza riga. Per verificare poi l'inattaccabilità della posizione lungo una diagonale, basta verificare che le regine non siano su rette di pendenza +1 o −1.

Con due regine, rispettivamente nelle posizioni X_1, Y_1 ed X_2, Y_2, basta verificare che

$$\frac{Y_1 - Y_2}{X_1 - X_2} \neq 1$$

cioè

$$|Y_1 - Y_2| - |X_2 - X_2| \neq 0$$

L'operazione è effettuata dalla funzione booleana *conflitto.*

Nella stampa dei risultati le caselle occupate da una regina sono indicate con *R*, le caselle vuote sono poste a zero.

Il programma che permette di ottenere tutte le configurazioni delle otto regine su una scacchiera è il seguente:

```
PROGRAM regine;
CONST
  n = 8;
VAR
  col: ARRAY[1 ... n] OF integer;
  nsol, riga, colonna, r: integer;
FUNCTION conflitto (c, r: integer): boolean;
VAR
  dr, k: integer;
BEGIN
  conflitto: = false;
  FOR k: = 1 TO c DO
    BEGIN
      dr: = abs(col[k] − r);
      IF dr *(dr − c − 1 + k) = 0
        THEN conflitto: = true;
    END;
END;
PROCEDURE soluzione (var sol: integer);
```

```
VAR
  j, k: integer;
BEGIN
  sol: = sol + 1;
  writeln ('soluzione no', sol);
  FOR j: = 1 TO n DO
    BEGIN
      FOR k: = 1 TO n DO
        BEGIN
          IF k = col[j]
             THEN write ('R')
             ELSE write ('0')
        END;
      writeln
    END;
END;

(*ricerca delle soluzioni*)

BEGIN
  nsol: = 0;
  FOR r: = 1 TO 8 DO
  BEGIN
    col[1]: = r; colonna: = 1; riga: = 1;
    WHILE colonna < > 0 DO
    BEGIN
      WHILE riga < = 8 DO
      BEGIN
        IF conflitto (colonna, riga)
          THEN riga: = riga + 1
          ELSE
          BEGIN
            col[colonna + 1]: = riga;
            colonna: = colonna + 1;
            IF colonna < = n - 1
               THEN riga: = 1
               ELSE
               BEGIN
                 soluzione (nsol);
                 riga: = n + 1
               END;
          END;
      END;

      WHILE (riga > 8) AND (colonna < > 8) DO
```

```
            BEGIN
              colonna: = colonna - 1;
              riga: = col [colonna + 1] + 1
            END;
        END;
      END;
    END.
```

Esecuzione

Qui diamo le prime soluzioni; se il programma è corretto, ne dovremmo stampare 92.

Soluzione 1

```
R 0 0 0 0 0 0 0
0 0 0 0 R 0 0 0
0 0 0 0 0 0 0 R
0 0 0 0 0 R 0 0
0 0 R 0 0 0 0 0
0 0 0 0 0 0 R 0
0 R 0 0 0 0 0 0
0 0 0 R 0 0 0 0
```

Soluzione 2

```
R 0 0 0 0 0 0 0
0 0 0 0 0 R 0 0
0 0 0 0 0 0 0 R
0 0 R 0 0 0 0 0
0 0 0 0 0 0 R 0
0 0 0 R 0 0 0 0
0 R 0 0 0 0 0 0
0 0 0 0 R 0 0 0
```

Soluzione 3

```
R 0 0 0 0 0 0 0
0 0 0 0 0 0 R 0
0 0 0 R 0 0 0 0
0 0 0 0 0 R 0 0
0 0 0 0 0 0 0 R
0 R 0 0 0 0 0 0
0 0 0 0 R 0 0 0
0 0 R 0 0 0 0 0
```

Soluzione 4

```
R 0 0 0 0 0 0 0
0 0 0 0 0 0 R 0
0 0 0 0 R 0 0 0
0 0 0 0 0 0 0 R
0 R 0 0 0 0 0 0
0 0 0 R 0 0 0 0
0 0 0 0 0 R 0 0
0 0 R 0 0 0 0 0
```

Soluzione 5

```
0 R 0 0 0 0 0 0
0 0 0 R 0 0 0 0
0 0 0 0 0 R 0 0
0 0 0 0 0 0 0 R
0 0 R 0 0 0 0 0
R 0 0 0 0 0 0 0
0 0 0 0 0 0 R 0
0 0 0 0 R 0 0 0
```

Soluzione 6

```
0 R 0 0 0 0 0 0
0 0 0 0 R 0 0 0
0 0 0 0 0 0 R 0
R 0 0 0 0 0 0 0
0 0 R 0 0 0 0 0
0 0 0 0 0 0 0 R
0 0 0 0 0 R 0 0
0 0 0 R 0 0 0 0
```

Soluzione 92

```
0  0  0  0  0  0  0  R
0  0  0  R  0  0  0  0
R  0  0  0  0  0  0  0
0  0  R  0  0  0  0  0
0  0  0  0  0  R  0  0
0  R  0  0  0  0  0  0
0  0  0  0  0  0  R  0
0  0  0  0  R  0  0  0
```

Come esercizio, si suggerisce di riscrivere questo programma utilizzando un algoritmo ricorsivo.

5 - TRATTAMENTO DELLE STRINGHE DI CARATTERI

Si è visto che le stringhe di caratteri possono essere considerate come vettori di caratteri. Si tenga presente inoltre che l'impiego dei vettori impaccati permette di risparmiare spazio di memoria.

Il tipo vettore impaccato può costituire il tipo *stringa* (*string* in inglese) su alcuni sistemi Pascal, e in particolare sul sistema dell'U.C.S.D.

Altri sistemi dispongono del tipo *alfa*, che è anch'esso un vettore impaccato, ma più limitato, in quanto corrisponde al numero dei caratteri di una parola-macchina: due se la parola è di sedici bits, quattro se è di trentadue bits.

Nella versione standard del linguaggio Pascal, non esiste una funzione standard che permetta di trattare le stringhe di caratteri; illustreremo pertanto delle procedure disponibili sul Pascal dell'U.C.S.D. e le procedure e le funzioni realizzabili su sistemi che ne siano sprovvisti.

5.1 - L'operazione di concatenazione

È un'operazione molto importante e nello stesso tempo semplicissima, necessaria nel trattamento delle stringhe di caratteri.

La *concatenazione* di due stringhe di caratteri consiste nel fondere le due stringhe iniziali in un'unica stringa, che si ottiene collocando i caratteri della seconda stringa immediatamente dopo quelli della prima.

Esempi

1) La concatenazione di "quand", di un apostrofo (') e di "anche" dà "quand'anche".

2) La concatenazione di "buona", di uno spazio e di "notte" dà la stringa "buona notte".

3) La concatenazione del prefisso "anti", di "costituzion" e della terminazione "almente" dà la stringa "anticostituzionalmente".

La concatenazione permette di ottenere stringhe di caratteri, costruendo così delle parole partendo da lettere o delle frasi partendo da parole.
Quest'operazione è alla base di qualsiasi linguaggio; in Pascal la concatenazione si può realizzare con una funzione o una procedura.
Sul sistema Pascal U.C.S.D. si usa una funzione che può venir richiesta ad esempio così:

```
s3 = concat(s1, s2);
```

Questa funzione può essere richiesta con più parametri - stringa:

```
concat(s1, s2 ...);
```

Esempi

1) Supponiamo di voler scrivere un programma che permetta di scrivere "buongiorno signora" o "buongiorno signore", a seconda che si risponda *f* o *m* ad una domanda relativa al sesso. Il programma sarà il seguente:

```
PROGRAM buongiorno;
CONST
   b = 'buongiorno';
VAR
   sb: string;
   sesso: char;
BEGIN
   readln (sesso);
   CASE sesso OF
      'f': sb: = concat(b,'signora');
      'm': sb: = concat(b,'signore')
   END;
   writeln (sb);
END.
```

2) Vediamo ora un altro programma:

```
PROGRAM stringhe;
CONST
   a = 'a';
   b = 'b';
TYPE
   stringa = string;
VAR
   s: stringa;
   j, l: integer;
BEGIN
   FOR l: = 1 TO 5 DO
   BEGIN
      s: = concat(s, a);
      FOR j: = 1 TO 2 DO
         BEGIN
            s: = concat (s, b);
            writeln (s);
         END;
   END;
END.
```

L'esecuzione del programma dà come risultato

```
ab
abb
abbab
abbabb
abbabbab
abbabbabb
abbabbabbab
abbabbabbabb
abbabbabbabbab
abbabbabbabbabb
```

Questa funzione è realizzabile, nella sua forma più semplice, con una procedura, in quanto il tipo *stringa*, non essendo considerato sempre come un tipo semplice, non è utilizzabile per definire una funzione:

```
PROCEDURE concat (VAR s: string; s1, s2: string);
CONST
  finestringa = '$';
VAR
  l, i: integer;
BEGIN
  i: = 1; l: = 1;
  WHILE s1[i] < > finestringa DO
    BEGIN
      s[l]: = s1[i];
      i: = i + 1; l: = l + 1
    END;
  i: = 1;
  WHILE s2[i] < > finestringa DO
    BEGIN
      s[l]: = l + 1; i: = i + 1
    END;
      s[l + 1]: = finestringa;
END;
```

5.2 - Funzioni e procedure per il trattamento delle stringhe di caratteri

Sono disponibili due tecniche per rappresentare, internamente al sistema, stringhe di caratteri: con la prima il contenuto della stringa è preceduto dalla lunghezza di quest'ultima; con la seconda il contenuto della stringa è seguito da un marcatore che ne indica la fine. Chiaramente, il marcatore non dev'essere un carattere che possa far parte della stringa.

Nel primo caso si ha questa struttura:

Lunghezza	CAR 1	CAR 2		CAR *n*...

Nel secondo caso la struttura sarà la seguente:

CAR 1	CAR 2		CAR *n*...	Marcatore fine stringa

La prima tecnica è utilizzabile a patto di fissare a priori la lunghezza massima della stringa. Si ha però il problema che il tipo della lunghezza della stringa e quello dei caratteri sono diversi, dal che derivano delle difficoltà di trattamento.

Nel secondo caso tutte le informazioni appartengono al medesimo tipo, ma è necessario escludere il carattere marcatore dall'insieme dei caratteri utilizzabili nelle stringhe.

5.2.1 - La funzione lenght

Questa funzione permette di conoscere la lunghezza di una stringa.

Nel Pascal U.C.S.D. prende il nome di *lenght* (cioè *lunghezza*) ed ha come parametro una variabile stringa.

La chiamata è molto semplice, ed ha la forma:

```
l: = lenght (s)
```

ove il parametro *s* è una stringa di caratteri (TYPE *string*).

Quando la funzione non è disponibile, può venir scritta facilmente. Ad esempio se definiamo il tipo *stringa*, scriveremo:

```
TYPE
   stringa: PACKED ARRAY[1 ... n] OF char;
FUNCTION lunghezza(s: stringa): integer;
CONST
   finestringa = '$';
VAR
   i: integer;
BEGIN
   i: = 1;
   WHILE s[i] < > finestringa DO
      i: = i + 1;
   lunghezza: = i − 1;
END;
```

5.2.1.1 - Un semplice programma di trattamento delle stringhe

Si tratta di un programma che permette di scrivere in senso inverso (da destra a sinistra) una stringa di caratteri.

```
PROGRAM stringhe;
TYPE
   stringa = string;
VAR
   s, si: stringa;
   l: integer;
PROCEDURE inversastringa (VAR si: stringa; s: stringa);
VAR
   k, l, i: integer;
BEGIN
   l: = lenght(s);
   k: = l; i: = 1;
   WHILE i <= l DO
```

```
        BEGIN
          si[i]: = s[k];
          i: = succ(i); k: = pred(k);
        END;
    END;
    BEGIN
      read (s);
      inversastringa (si, s);
      writeln (si);
    END.
```

Come risultato dell'esecuzione si ha:

```
    esempio d'inversione (R)
    enoisrevni'd oipmese
```

5.2.1.2 - Esercizi sul trattamento delle stringhe

1. Scrivere un programma Pascal che cerchi se un carattere è presente in una stringa di caratteri.
2. Scrivere un programma che calcoli la frequenza con cui un carattere compare in una stringa.
3. Scrivere un programma che, sulla base dell'esempio del paragrafo precedente, determini se una parola, o una frase, sono palindrome, cioé se costituiscono delle stringhe di caratteri identiche nei due sensi.

Soluzioni

1. e 2. Il programma che segue risponde ai primi due esercizi.

```
    PROGRAM stringhe;
    TYPE
      stringa = string;
    VAR
      s: stringa;
      I: integer;
      car: char;
    FUNCTION freqcar(s: stringa; car: char): integer;
    VAR
      I, f: integer;
    BEGIN
      f: = 0;
      FOR I: = 1 TO lenght(s) DO
        IF s[I] = car THEN f: = f + 1;
      freqcar: = f;
    END;
```

```
    BEGIN
      readln(s);
      readln(car);
      writeln('il carattere', car,
        'compare', freqcar(s, car),
        'volte in', s);
  END.
```

Esecuzione

```
  Mastro don Gesualdo
  o Ⓡ
  il carattere o compare 3 volte in Mastro don Gesualdo
```

3.

```
  PROGRAM palindroma;
  TYPE
    stringa: string;
  VAR
    s, si: stringa;
    l: integer;
  PROCEDURE inversastringa (VAR si: stringa; s: stringa);
  VAR
    k, l: integer;
  BEGIN
    l: = lenght(s); si: ' ';
    FOR k: = l DOWNTO 1 DO
      si: = concat(si, copy(s, k, 1) );
  END;
  BEGIN
    read(s);
    inversastringa(si, s);
    writeln(lenght(si) );
    writeln(si);
    IF s = si THEN
      writeln (s, 'è' 'palindroma');
  END.
```

5.2.2 - Le funzioni di estrazione di stringhe

L'operazione inversa della concatenazione è l'estrazione di una sottostringa da una stringa data.

In Pascal U.C.S.D. questa funzione ha il nome di *copy*. I parametri sono i seguenti: stringa di partenza, posizione e lunghezza della sottostringa da estrarre.

La chiamata di questa funzione ha la forma

copy(stringasorgente, posinizio, dimensione)

Stringasorgente è la stringa di partenza; *posinizio* e *dimensione* sono parametri di tipo *intero* che specificano la posizione d'inizio (cioè il numero d'ordine del primo carattere) e la lunghezza della sottostringa da estrarre.

Anche questa funzione si scrive facilmente in forma di procedura:

```
PROCEDURE sottostringa  (s: string; ss: string;
                         pos, lun: integer);
CONST
  finestringa = '$';
VAR
  n, i: integer;
BEGIN
  n: = lunghezza(s);
  IF n > = pos THEN
    BEGIN
      FOR i: = 1 TO lun DO
        ss[i]: = s [pos + i - 1];
        ss[lun + 1]: = finestringa;
    END
    ELSE ss[1]: = finestringa;
END.
```

5.2.3 - Altre funzioni di trattamento delle stringhe

Con le funzioni e le procedure or ora definite si possono effettuare tutte le operazioni di trattamento delle stringhe di caratteri.

In ogni caso esistono delle altre procedure e funzioni che sono particolarmente utili nell'elaborazione di testi, quali soprattutto le funzioni (e procedure) di cancellazione ed inserimento di stringhe in un'altra stringa.

Nel Pascal U.C.S.D. queste funzioni sono disponibili, e si chiamano rispettivamente

delete (cioè cancellare)

e

insert (cioè inserire)

La richiesta della funzione *cancellare* ha la forma:

delete (stringa, posizione, dimensione)

ove
- il parametro *stringa* definisce la stringa di caratteri interessata dalla cancellazione;
- il parametro *posizione* è un indice di tipo *intero* che specifica l'inizio della sottostringa da cancellare;
- il parametro *dimensione* indica la lunghezza in caratteri della sottostringa da cancellare.

La funzione d'*inserimento* viene richiesta come segue:

```
insert(stringapartenza, stringadestinazione, posizione)
```

Qui
- il parametro *stringapartenza* contiene la stringa da inserire;
- il parametro *stringadestinazione* indica la stringa nella quale va effettuato l'inserimento;
- il parametro *posizione* è un indice *intero* che specifica la posizione a partire dalla quale si deve effettuare l'inserimento nella stringa di arrivo.

C'è poi un'altra funzione che provvede ad individuare la *posizione* di una sequenza di caratteri entro un'altra stringa. Questa funzione si chiama *pos*, ed ha come parametri di richiesta due stringhe di caratteri:

```
pos (stringaimmagine, stringasorgente)
```

- Il primo parametro, *stringaimmagine*, è la stringa che si cerca;
- il secondo, *stringasorgente*, specifica la stringa entro la quale si deve ricercare l'esistenza e la posizione della prima stringa.

La posizione è quindi ritornata come risposta quando si chiama la funzione *pos*, che è una funzione di tipo *intero*.

5.2.4 - La funzione di conversione di un numero in una stringa di caratteri

Questa funzione, che è particolare del sistema U.C.S.D., permette di trasformare il contenuto di una variabile numerica in una stringa di caratteri formata dalle cifre corrispondenti al valore della variabile.

La procedura ha questa sequenza di richiesta:

```
str (numero, stringacifra);
```

Il primo parametro è una variabile di tipo *intero*, il secondo una variabile di tipo *stringa* di caratteri.

5.2.5 - Applicazioni

1. Ricerca di una parola in una stringa e calcolo della frequenza con cui compare. Ricorreremo qui alla procedura *copy*; di conseguenza il programma cercherà un'immagine della parola che si vuol trovare, anche se l'immagine fa parte di una stringa più lunga.

```
PROGRAM stringhe;
TYPE
   stringa = string;
VAR
   s: stringa;
   l: integer;
   car: char;
   parola: stringa;
PROCEDURE ricercaparola(VAR n: integer; s, parola: stringa);
VAR
   i, k, l: integer;
BEGIN
   l: = lenght(s); k: = lenght (parola);
   n: = 0;
   FOR i: = 1 TO l DO
      BEGIN
         IF s[i] = parola[1] THEN
            IF parola = copy(s, i, k) THEN
               n: = n + 1;
      END;
END;
FUNCTION freqcar(s: stringa; car: char): integer;
VAR
   l, f: integer;
BEGIN
   f: = 0;
   FOR l: = 1 TO lenght(s) DO
      IF s[l] = car THEN f: = f + 1;
   freqcar: = f;
END;
BEGIN
   readln(s);
   readln(parola);
   ricerca(l, s, parola);
   writeln ('la parola', parola,
   'compare', l, 'volte in', s);
END.
```

2. Elaborazione di testi
   Il programma che segue permette di ottenere la posizione di una parola in una stringa, come pure di cancellare, inserire e rimpiazzare una parola in un testo. È quindi un piccolo programma di elaborazione di un testo.

```
PROGRAM elabtesto;
TYPE
  stringa = string;
VAR
  parola, immagine, s: stringa;
  p, l: integer;
  car: char;
BEGIN
  writeln ('scrivi un testo');
  readln(s);
  s: = concat(s, '.');
  car: = 'p';
  writeln ('p(osizione, c(ancellare, i(nserire, r(impiazzare');
  WHILE car IN['p', 'c', 'i', 'r'] DO
    BEGIN
      readln (car);
      CASE car OF
        'p': BEGIN
               write ('posizione di?');
               readln (immagine);
               writeln (pos(immagine, s) );
             END;
        'c': BEGIN
               write ('cancellare?');
               readln (immagine);
               l: = lenght(immagine);
               delete(s, pos(immagine, s), l);
               writeln (s);
             END;
        'i': BEGIN
               write ('inserire?');
               readln (immagine);
               write ('a partire da?');
               readln (parola);
               insert (immagine, s, pos(parola, s) );
               writeln (s);
             END;
```

```
          'r': BEGIN
                  write ('rimpiazzare che cosa?');
                  readln (immagine);
                  p: = pos(immagine, s);
                  delete(s, p, lenght(immagine) );
                  write ('con che cosa?');
                  readln (immagine);
                  insert(immagine, s, p);
                  writeln (s);
               END;
      END;
   END;
END.
```

Esempio di esecuzione

```
scrivi un testo
Cuesta mattina l'aradio a' trasmesso una canzone di Montan. Ⓡ
p(osizione, c(ancellare, i(nserire, r(impiazzare
p Ⓡ
posizione di? una Ⓡ
38
r Ⓡ
rimpiazzare che cosa? c Ⓡ
con che cosa? q Ⓡ
questa mattina l'aradio a' trasmesso una canzone di Montan.
c Ⓡ
cancellare? l'a Ⓡ
questa mattina radio a' trasmesso una canzone di Montan.
i Ⓡ
inserire? la Ⓡ
a partire da? ra Ⓡ
questa mattina la radio a' trasmesso una canzone di Montan.
c Ⓡ
cancellare? a' Ⓡ
questa mattina la radio trasmesso una canzone di Montan.
i Ⓡ
inserire? ha Ⓡ
a partire da? tra Ⓡ
questa mattina la radio ha trasmesso una canzone di Montan.
i Ⓡ
inserire? d Ⓡ
a partire da? . Ⓡ
questa mattina la radio ha trasmesso una canzone di Montand.
```

5.3 - Funzioni e procedure per il trattamento di vettori di caratteri

Il Pascal U.C.S.D. dispone di un certo numero di funzioni e di procedure che permettono di lavorare sui vettori di caratteri.

Queste procedure non sono standard, dipendono spesso dai sistemi sui quali operano. In ogni caso è chiaro che, utilizzandole, occorre rispettare scrupolosamente l'elenco dei parametri ed i tipi.

Si tratta di funzioni e procedure che permettono di esaminare un vettore di caratteri, di spostare blocchi di caratteri e di riempire rapidamente, e completamente, un vettore con un determinato carattere.

5.3.1 - La funzione di scansione

Questa procedura, detta *scan* (cioè *scansione*), esamina un vettore nei due sensi e fornisce il numero di caratteri considerati finchè non si verifichi una certa condizione o finchè la scansione non arrivi al termine.

Questa funzione viene richiesta come segue:

scan(lunghezza, espressione, vettore)

ed è di tipo *intero*. Il parametro *vettore* indica il nome del *vettore di caratteri impaccati* o l'elemento a partire dal quale deve cominciare la scansione.

Il parametro *lunghezza* è un intero che indica il numero di caratteri da esaminare a partire dalla posizione iniziale: può essere positivo o negativo, per cui il vettore può venir scandito nei due sensi (avanti e indietro).

Il parametro *espressione* indica una condizione di arresto della scansione. Il suo tipo è: operatore = o < > seguito da un'espressione di tipo carattere.

Esempio

Si abbia il vettore di caratteri *stringa*, contenente questa sequenza di caratteri:

'un lancio dei dadi ... non eliminerà mai il caso'

La funzione *scan* darà, ad esempio, i seguenti valori:

scan(30,='.',stringa) uguale a 18

oppure

scan(–20, < > '.', stringa [20]) uguale a 17

5.3.2 - Le procedure di spostamento di vettori di caratteri

Esistono due diverse procedure, a seconda che si debba spostare un blocco di caratteri partendo da destra o da sinistra.

Il modo di richiesta ed i parametri delle due procedure sono identici.

Partendo da sinistra, abbiamo:

```
moveleft (sorgente, destinazione, lunghezza)
```

Questa procedura sposta un numero di caratteri uguale al parametro *lunghezza* dal vettore *sorgente* al vettore *destinazione*: lo spostamento avviene da sinistra verso destra.

Partendo da destra invece si ha:

```
moveright (sorgente, destinazione, lunghezza)
```

Il significato dei parametri è lo stesso, ma questa volta la posizione di partenza è a destra e lo spostamento nei due vettori avviene da destra verso sinistra.
Attenzione: se i vettori *sorgente* e *destinazione* coincidono, bisogna stare attenti a non distruggere il vettore sorgente durante lo spostamento. La scelta della procedura da usare dipende da come sono collocate reciprocamente le due parti sorgente e destinazione.

5.3.3 - Le procedure di riempimento di un vettore di caratteri

Si tratta della procedura *fillchar* (cioè riempimento di caratteri), la cui sequenza di richiesta è:

```
fillchar (destinazione, lunghezza, carattere)
```

Il parametro *lunghezza* indica il numero di caratteri uguali, definiti dal parametro *carattere*, con i quali va riempito il vettore *destinazione*.

5.3.4 - La funzione dimensione di una variabile

Questa funzione non è specifica del trattamento dei vettori di caratteri, ma è molto utile in quest'operazione.

Il suo parametro è un identificatore di variabile, o di tipo, che fornisce il numero di bytes occupati dalla variabile o dal tipo in questione.

La sequenza di richiesta è:

```
sizeof (identificatore)
```

1. Calcolo della radice quadrata di *a*.
   Calcolare $\sqrt{a}$ con la formula di ricorrenza

$$x_{n+1} = \frac{1}{2} \left(x_n + \frac{a}{x_n}\right)$$

   e sapendo che $x_0 = \frac{(1 + a)}{2}$ (metodo di Newton).

   Le iterazioni si arrestano quando

$$\frac{x_{n+1} - x_n}{x_n} > \epsilon = 10^{-5}$$

2. Calcolare la radice cubica di *a*.
   Come per $\sqrt{a}$, useremo il metodo di Newton:

$$x_{n+1} = x_n + \frac{1}{3} \left(\frac{a}{x_n^2} - x_n\right)$$

3. Calcolo di una radice di $f(x). = 0$. con il metodo della dicotomia o bisezione: se $f(a) \cdot f(b) < 0$, una radice di $f(x) = 0$ è compresa fra *a* e *b*.
   È possibile calcolare il valore di questa radice. Per verificare se la radice è compresa fra *a* ed *m* (valore medio di *a*, *b*), o fra *m* e *b*, si reitera la bisezione fino a trovare una radice posta fra due valori la cui differenza sia minore della precisione voluta.
4. Integrazione numerica.
   Si debba integrare la seguente funzione di *x*:

$$\Phi(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

   per $0 \leq x < \infty$
   L'approssimazione di Hastings porta all'espressione seguente:

$$\Phi(x) = 1 - \frac{1}{(1 + a_1x + a_2x^2 + a_3x^3 + a_4x^4)^4}$$

I coefficienti a_1, a_2, a_3 ed a_4 hanno i seguenti valori:

$$a_1 = 0.278393$$
$$a_2 = 0.230389$$
$$a_3 = 0.000972$$
$$a_4 = 0.078108$$

Scrivere una funzione che permetta di calcolare questo integrale.

5. Integrazione numerica: metodo dei trapezi.
Si abbia una funzione $y = f(x)$. Sull'intervallo $[a, b]$ l'integrale

$$A = \int_a^b f(x)dx$$

rappresenta l'area compresa fra la curva espressa da $f(x)$, l'asse delle ascisse e le verticali d'ascissa a e b. Scomponendo l'area $[a, b]$ in segmenti di lunghezza dx, l'area A può essere definita come uguale alla somma delle aree parziali dA.
Per dx sufficientemente piccolo, dA può essere calcolata con la formula seguente:

$$dA = \frac{f(x_{i+1}) + f(x_i)}{2}$$

Scrivere una funzione che permetta di calcolare A.

6. Calcolare $I = \int_a^b f(x)dx$ con il metodo dei rettangoli, secondo la formula

$$I = \frac{b - a}{n} \sum_{i=0}^{n-1} f\left(a + i\,\frac{b - a}{n}\right)$$

ove n e il numero d'intervalli adottato.
Scrivere una funzione che permetta di calcolare I.

7. Scrivere un programma di conversione di un numero decimale, in una base qualunque ≤ 10.

8. Scrivere un programma di conversione da decimale in esadecimale (base 16). I numeri da 0 a 15 sono rappresentati dalle lettere A, B, C, D, E, F (cifre esadecimali).

9. Scrivere un programma di conversione di un numero a base qualunque ($\leq$ 16) in un numero decimale.

10. Il sistema di numerazione romano utilizza i simboli *M*, *D*, *C*, *L*, *X*, *V*, *I* per rappresentare i numeri 1000, 500, 100, 50, 10, 5, 1. In romano antico un numero era espresso con una sequenza di tali simboli, scritti da sinistra a destra. Il valore del numero era dato dalla somma dei valori relativi a ciascun simbolo. In romano moderno i simboli *C*, *X*, *I* possono precedere un altro simbolo di valore maggiore, dal quale vanno detratti. Ad esempio,

Decimale	Romano antico	Romano moderno
4	IIII	IV
9	VIIII	IX
91	LXXXXI	XCI

    a) Scrivere un programma che legga un numero in romano antico e stampi il valore decimale corrispondente. Fare poi l'operazione inversa, cioè da decimale a romano antico.

    b) Eseguire una somma in romano antico, senza passare per il valore decimale, e stampare il risultato in romano antico.

    c) Effettuare la stessa operazione passando per il valore decimale, e stampare il risultato in romano antico.

11. Scrivere un programma che effettui le seguenti operazioni:

    a) lettura in romano moderno, conversione e stampa in decimale;

    b) lettura di un numero decimale, stampa in romano moderno;

    c) somma, sottrazione e moltiplicazione di due numeri in romano moderno, e stampa del risultato sempre in romano moderno.

12. Scrivere un programma che legga un numero decimale e lo stampi in lettere. Ad esempio, leggendo 1291, verrà stampato "milleduecentonovantuno". Il programma dev'essere valido con numeri di sette cifre al massimo.

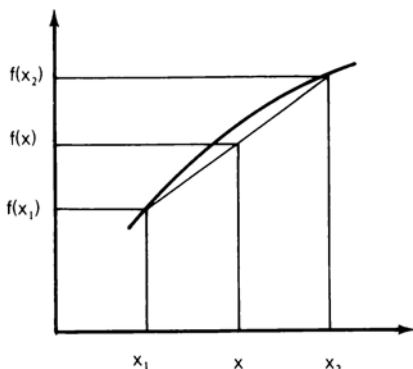
13. Scrivere un programma che legga un numero in lettere e lo stampi in decimale. L'operazione è inversa alla precedente.

14. Scrivere un programma che stampi un calendario, tenendo conto degli anni bisestili e del fatto che gli anni iniziali di ogni secolo (1800, 1900, 2000, etc.) non sono mai bisestili. Gli anni dovranno essere stampati in lettere. Come esempio si possono prendere gli anni 1978, 1980 e 2000.

15. Scrivere un programma che calcoli x^2 per

$$x = 1, 1.1, 1.2, 1.3, \dots 9.8, 9.9, 10$$

I valori di x devono essere disposti in un vettore X, i valori di x^2 in un vettore $X2$.

16. Scrivere, riferendosi ad un metodo d'interpolazione lineare, un programma che permetta di calcolare i valori approssimati di $\sqrt{1}$, $\sqrt{2} \dots \sqrt{100}$.

Si ricordi che l'interpolazione lineare si basa sul seguente principio:
se abbiamo

$$x_1 < x < x_2$$

e sono noti i punti

$$\begin{cases} x_1 \\ f(x_1) \end{cases} \qquad \begin{cases} x_2 \\ f(x_2) \end{cases}$$

il valore $f(x)$ si ottiene prendendo il valore approssimato corrispondente alla retta che congiunge i due punti noti. Si ha quindi:

$$\frac{f(x_2) - f(x_1)}{f(x) - f(x_1)} = \frac{x_2 - x_1}{x - x_1}$$

17. Si abbia un'equazione della forma

$$f(x) = a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = 0$$

L'equazione rappresenta un polinomio di sesto grado in cui uno qualunque dei coefficienti può essere nullo.
Se a_0, a_1 ... a_6 sono dati, scrivere un programma che legga questi dati, calcoli $f(x_0)$ e stampi i coefficienti, x_0 ed $f(x_0)$ per x_0 che varia da -10 a $+10$ con un incremento di 0.1.

18. Considerare la sequenza

1. *e* assume il valore 1;
2. *d* assume il valore 5;
3. sostituire *e* con $1+(e.x/d)$;
4. se $d = 1$ stop. Sennò sottrarre 1 da *d* e tornare al punto 3.

Al termine dell'algoritmo si ha:

$$e = 1 + x + \frac{x^2}{2} + \frac{x^3}{2.3} + \frac{x^4}{2.3.4} + \frac{x^5}{2.3.4.5} \simeq e^x$$

Scrivere il relativo programma supponendo che *x* sia letto come dato.

19. Applicare il metodo esposto nell'Esercizio 9 a

$$ctg(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9}$$

e scrivere il programma relativo.

20. Come si può calcolare 2^{200} in modo esatto benchè il numero sia molto grande?

21. Il minimo comune multiplo è il più piccolo intero divisibile per *a* e *b*.
Scrivere il programma che permette di calcolare il minimo comune multiplo di due interi *a* e *b*.

22. *A* è un vettore (20, 20). Scrivere un programma che calcoli la somma dei valori assoluti degli elementi della *k-esima* riga di *A*, ad esclusione di *A*(*k*, *k*).

$$\text{som abs} = \sum_{j \neq k} |A_{kj}|$$

23. Si abbia un campione di *n* valori, dei quali se ne debbano selezionare alcunï.

a) Vogliamo selezionare i valori compresi fra $m - 2\sigma$ ed $m + 2\sigma$, sapendo che *m* è la media degli *n* valori, e σ è lo scarto tipico. σ è calcolato come segue:

$$\sigma = \sqrt{\sum_{i=1}^{n} (x_i - m)^2 \, p(X = x_i)}$$

in cui se x_i è un elemento del vettore, assumeremo che

$$p(X = x_i) = \frac{1}{n}$$

b) I valori selezionati vengono messi in un secondo vettore. Ci proponiamo ora di selezionare gli x_i del secondo vettore per i quali

$$m - \frac{m_i - 1}{2} \leq x_i \leq m \frac{m_i - 1}{2}$$

ove

$$m_{i-1} = \frac{1}{i-1} \sum_{i=1}^{i-1} xj$$

c) Scrivere il programma che esegue le seguenti operazioni:

- Leggere un vettore *A* di 50 elementi e stamparlo.
- Effettuare la prima selezione sul vettore *A* e trasferire i valori così scelti in un vettore *B*. Stampare *B*.
- Effettuare la seconda selezione sul vettore *B*. I valori cosi ottenuti saranno trasferiti in un vettore *C*. Stampare *C*.
- Ordinare i valori di *C* e stampare il vettore così ordinato.

24. Il gioco del Nim.
Il gioco del Nim, comunemente noto come Marienbad, è fatto qui con 12 fiammiferi, disposti su tre file in questo modo:

Il gioco consiste nel togliere dei fiammiferi secondo alcune regole, che daremo fra poco. Vince il giocatore che per ultimo toglie uno o più fiammiferi.
L'insieme formato da n_1 fiammiferi nella prima fila, n_2 nella seconda ed n_3 nella terza può essere espresso con la terna (n_1, n_2, n_3). Qui l'insieme di partenza del gioco è dato dalla terna (3, 4, 5). I giocatori sono due, e ciascuno alternativamente può togliere uno o più fiammiferi appartenenti ad una stessa fila. Quello che toglie gli ultimi fiammiferi vince (situazione 0, 0, 0).
Chiamiamo i due giocatori x ed y. Una situazione (n_1, n_2, n_3) raggiunta da x è detta vincente se, qualunque sia la prima mossa di y, esiste per x una mossa tale da portarlo alla situazione (0, 0, 0). Il giocatore x, una volta raggiunta una situazione vincente, può procedere in modo sistematico verso la vittoria contrapponendo ad una qualsiasi mossa dell'avversario y una mossa che lo porti di nuovo ad una situazione vincente. Questa strategia dà al giocatore x la possibilità di raggiungere la situazione (0, 0, 0) e di vincere così la partita. È quindi chiaro che la conoscenza delle combinazioni vincenti è di un'importanza fondamentale.
Nel gioco del Nim infatti si può benissimo determinare se una combinazione è o no vincente: l'algoritmo è applicabile ad un numero qualsivoglia di fiammiferi e di file, e con una qualunque disposizione iniziale.

1. Sia $b_1b_2b_3$ l'equivalente binario di n_1; $c_1c_2c_3$ quello di n_2 e $d_1d_2d_3$ quello di n_3.

2. Sia

$$r_1 = b_1 + c_1 + d_1$$
$$r_2 = b_2 + c_2 + d_2$$
$$r_3 = b_3 + c_3 + d_3$$

3. Se r_1, r_2 ed r_3 sono tutti pari, allora (n_1, n_2, n_3) è una combinazione vincente, sennò è perdente.
   Ad esempio, (3, 4, 5) non è una combinazione vincente.
   In binario,

   3 è $b_1b_2b_3 = 011$
   4 è $c_1c_2c_3 = 100$
   5 è $d_1d_2d_3 = 101$

   ed $r_2 = 1 + 0 + 0$ è dispari,

Invece (1, 4, 5) è una combinazione vincente.
In binario,

$$1 \text{ è } b_1b_2b_3 = 001$$
$$4 \text{ è } c_1c_2c_3 = 100$$
$$5 \text{ è } d_1d_2d_3 = 101$$

e

$$r_1 = 0 + 1 + 1 = 2$$
$$r_2 = 0 + 0 + 0 = 0$$
$$r_3 = 1 + 0 + 1 = 2$$

sono tutti pari.

Scrivere un programma che simuli delle partite a Nim con un giocatore *a* contrapposto ad un giocatore *b* (che sarà il calcolatore, o meglio il programma): il giocatore *a* fa la sua mossa introducendo il numero dei fiammiferi *l* da togliere dalla sua fila *m*. Scrivete prima un programma nel quale il giocatore *b* scelga a caso fra le diverse mosse possibili, e poi un altro programma nel quale il giocatore *b* cercherà di realizzare una combinazione vincente. In questo caso, se non potrà farlo, toglierà un fiammifero dalla prima fila possibile. Vedrete che *b* vincerà tutte le volte che avrà la prima mossa, quando la combinazione di partenza è vincente.

25. Il gioco del bridge (semplificato).
Il bridge è un gioco di carte relativamente recente: nato nel 1896, è una derivazione del whist (inglese).
Si gioca con cinquantadue carte, fra due coppie di giocatori.
Scopo del gioco è fare il maggior numero di prese possibile, non scendendo al di sotto del numero di prese dichiarato nel contratto iniziale.
Si comincia distribuendo tredici carte per giocatore. C'è poi una serie di dichiarazioni che si conclude normalmente con una dichiarazione massima fatta da una delle due coppie, che s'impegna così a realizzare (come minimo) il corrispondente numero di prese. Ad esempio, se la dichiarazione è di 1 *fiori*, la coppia s'impegna a fare 6 + 1 = 7 prese, con il colore *fiori* come atout.

Valore delle carte
- Il valore delle carte è in ordine decrescente: asso, re, dama, fante, 10, 9 ... 2.
- I relativi punti sono: asso = 4, re = 3, dama = 2, fante = 1.

- I colori hanno valori diversi che, in ordine crescente, sono: fiori, quadri, cuori, picche. Di conseguenza le dichiarazioni si fanno secondo l'ordine crescente dei valori: un fiori, un quadri, un cuori, un picche, due fiori, e così via.

Non considereremo qui la dichiarazione senza atout.

Il meccanismo della dichiarazione

Ciascun giocatore ha la possibilità di passare o di far salire le dichiarazioni. Ad esempio, se un giocatore ha dichiarato un *cuori*, quello che gioca dopo di lui può passare o dichiarare un *picche*, o due *fiori*, o più. La dichiarazione di un giocatore è determinata dal numero dei punti di cui quel giocatore dispone. Valgono le seguenti convenzioni:

- Al primo giro di dichiarazioni per poter dichiarare bisogna avere almeno 13 punti e si dichiarerà *uno* nel colore più lungo. Se si hanno più di 24 punti, si può dichiarare *due* nel colore più lungo.
- Se il compagno ha aperto, e gli avversari non hanno rilanciato, il giocatore, se ha meno di 6 punti, passa; se ne ha da 7 a 10, dichiarerà una presa in più nel colore con cui ha aperto il compagno; se ne ha 10 o più, farà una dichiarazione nel colore più alto che ha in mano.
- Se il compagno ha aperto ed uno degli avversari ha rilanciato, con 9 punti o meno di 9 il giocatore passa, con 10 o più dichiara una presa in più nel colore con cui il compagno ha aperto.

a) Scrivere un programma in Pascal che calcoli, per una mano di tredici carte, il numero dei punti corrispondente al valore di queste carte e la ripartizione delle carte nei vari colori. Dopo aver messo a punto il programma, trasformatelo in una procedura in cui non ci siano istruzioni d'ingresso/uscita.
b) Scrivere un programma principale con cui leggere e stampare un insieme di cinquantadue carte distribuite a caso, conteggiare i punti, aggiungere i punti di colore per ricavare il punteggio totale e mettere in ordine le carte per colore.
   I risultati saranno visualizzati nella forma seguente:

Giocatore 1

picche	*cuori*	*quadri*	*fiori*	*punti*	*punti di colore*	*totale*
d	a	9	a	13	2	15
f	d	7				
10	5	6				
9		3				
8						

Giocatore 2

c) Completare il programma precedente in modo da avere le dichiarazioni relative alle quattro mani precedenti.
d) Per generare delle sequenze casuali con le quali ottenere dei dati pure casuali, utilizzeremo la funzione *random* (x), che fornirà una sequenza di numeri pseudocasuali (v. Cap. 9). Scrivere una procedura che generi una distribuzione delle carte pseudocasuale, e con questa procedura rieseguire il punto c) per un massimo di cinque distribuzioni diverse.

OSSERVAZIONI

1. Nei punti b) e c) il programma principale legge tutte le cinquantadue carte, distribuite in ordine casuale, nel seguente modo:

   7. *picche*, 8. *quadri*, a. *cuori*, f. *cuori*, 10. *fiori*,
   2. *quadri*, d. *picche*.

   Quindi è il programma che "dà le carte": il giocatore 1 avrà il 7 di *picche*, il 10 di *fiori*, etc. il giocatore 2 l'8 di *quadri*, il 2 di *quadri*, e così via.

2. Nel punto d) la sequenza di carte non è più letta, ma è generata con un processo casuale.

CAPITOLO 6

I RECORDS ED I FLUSSI

> *"L'uomo non è che una canna, la più debole della natura; ma è una canna pensante... Tutta la nostra dignità consiste dunque nel pensiero. È con esso che dobbiamo confrontarci, non con lo spazio e con la durata, che non saremmo in grado di colmare. Adoperiamoci quindi a ben pensare: è questo il fondamento della morale".*
>
> PASCAL,
> Pensées

In questo capitolo introdurremo delle nuove strutture di dati, dette records.

Il concetto di record è abitualmente associato a quello di flusso, pertanto tratteremo qui anche dei flussi.

Va detto comunque che in Pascal il record è un tipo strutturato che può essere indipendente dal concetto di flusso: un flusso Pascal può non essere strutturato in records; un record può non essere associato ad un flusso.

Nella prima parte del capitolo esamineremo le strutture di tipo record, nella seconda parte i flussi.

1 - LA STRUTTURA DI TIPO RECORD

Un flusso è generalmente formato da records; ad un insieme di records si può accedere con diversi metodi, sequenziali o diretti.

In Pascal un record è definibile per mezzo di una dichiarazione, ma *non è obbligatoriamente associato ad un flusso.*

Un record serve semplicemente a raggruppare un insieme di dati di tipi diversi sotto un generico nome comune.

Un indirizzo, o una data, possono ad esempio essere considerati dei records, ma possiamo definire in forma di record anche un oggetto matematico, come ad esempio un numero complesso o delle coordinate cartesiane.

1.1 - Il concetto di campo

Un record è definito da un insieme di dati componenti, che prendono il nome di *campi*. I campi sono dati elementari che possono essere di diversi tipi.

Esempi

1) Un indirizzo è scomponibile in sei campi:
   - il campo cognome, alfabetico;
   - il campo nome, alfabetico;
   - il campo via, alfanumerico;
   - il campo codice postale, numerico;
   - il campo città, alfabetico o scalare;
   - il campo nazione, alfabetico o scalare.

   In Pascal questo record sarà definito dalle seguenti dichiarazioni:

```
TYPE
  codicepostale = RECORD provincia: 0 ... 99
                         località: 0 ... 999
                  END;
alfa: PACKED ARRAY[1 ... 40] OF char;
indirizzo = RECORD
              cognome: alfa;
              nome: alfa;
              via: alfa;
              codice: codicepostale;
              citta: alfa;
              nazione: (Austria, Belgio, Danimarca, Francia, Germania, Gran
                        Bretagna, Irlanda, Italia, Lussemburgo, Norvegia, Paesi
                        Bassi, Portogallo, Spagna, Svezia, Svizzera, Ungheria,
                        U.R.S.S., U.S.A.)
            END;
```

2) Definizione di un punto di coordinate x (ascissa) ed y (ordinata).

```
TYPE
  punto = RECORD ascissa: real;
                 ordinata: real
          END;
```

3) Definizione di un orario.

```
orario = RECORD ora: 0... 23
                minuto: 0... 59;
                secondo: 0 ... 59
         END;
```

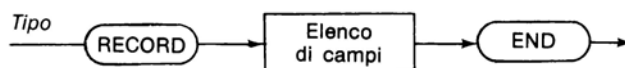
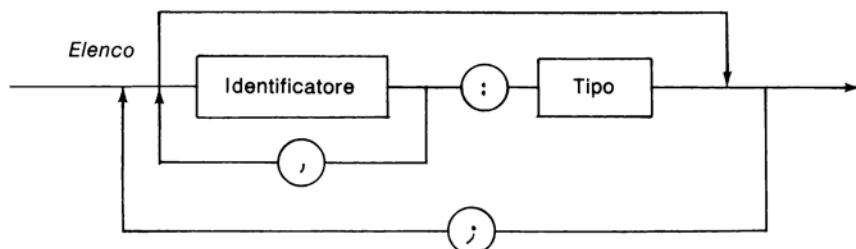
4) La struttura record è un tipo, e di conseguenza possiamo annoverarla fra i tipi di variabile. Una curva può essere definita come un vettore di punti:

```
VAR
  curva: ARRAY [1 ... 100] OF punto;
```

5) Analogamente, il percorso di un treno con un orario di passaggio per varie città può essere definito come segue:

```
TYPE
  citta = (Roma, Firenze, Bologna, Milano);
VAR
  Settebello: ARRAY [citta] OF orario;
```

1.2 - La struttura sintattica di un record

Un record è definito da una dichiarazione di tipo formata da un elenco di campi preceduto dalla parola riservata RECORD e seguito dalla parola riservata END. Quindi basta aggiungere nel diagramma sintattico dei tipi la seguente struttura:

Ne abbiamo appena visto degli esempi. La struttura di *campo*, così come l'abbiamo illustrata, è definita da

Ogni record comprende, dunque, un elenco di dati, simile per tutti i records. In alcuni casi però è utile lavorare con records i cui dati hanno strutture diverse in funzione del contenuto di una variabile.

Questi *campi varianti* possono essere formati da un numero anch'esso variante di sottocampi, di tipo diverso secondo i casi. Ciascun campo variante può dunque essere formato da un elenco di sottocampi, definiti fra parentesi per ciascuna alternativa possibile. Le varie alternative sono definite dal valore di un parametro specificato per mezzo di una struttura CASE: si noti che qui non si tratta di un'istruzione eseguibile, ma di una *dichiarazione* che permette di specificare delle strutture di dati che variano in funzione del valore di un altro dato.

Esempio

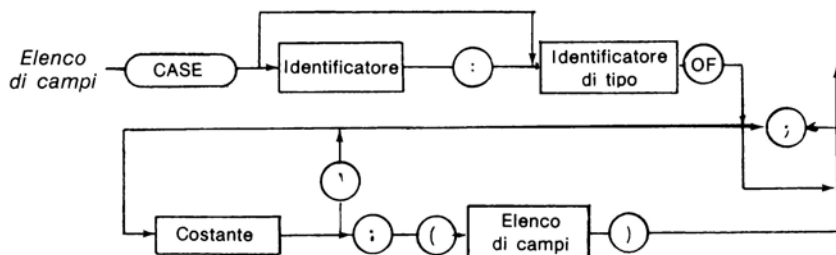
Supponiamo di dover definire un record destinato a contenere le informazioni relative alle iscrizioni degli studenti a delle materie, o corsi, complementari, regolate da differenti modalità di controllo.

La struttura dei records con varianti può essere enunciata, per esteso, in questo modo:

- se la materia è letteraria, il controllo è formato da un voto per lo scritto ed un voto per l'orale;
- se la materia è una lingua viva, il controllo è formato da un voto in composizione, un voto per le traduzioni ed infine un voto per l'orale;
- se la materia è matematica, il controllo è formato da un voto in analisi, uno in algebra ed uno in logica;
- se la materia è informatica, il controllo è formato da un voto in programmazione, uno in elettronica digitale ed uno in teoria degli automi e dei linguaggi.

Il record relativo può venir strutturato in questo modo:

```
TYPE
  studente = RECORD
  ...
    (*parte comune a tutti gli studenti*)
    CASE materia OF
      letteratura: (*modalità controllo let.*)
      lingua: (*modalità controllo l. v.*)
      matematica: (*modalità controllo mat.*)
      informatica: (*modalità controllo info.*)
            END
```

Prima di scrivere l'esempio in una forma più complessa, è opportuno precisare la sintassi dei campi varianti di un record. Il diagramma è il seguente:

La definizione è ricorsiva, in quanto un elenco di campi può comprendere altri e-lenchi di campi: questo permette di avere strutture di records varianti, strutturate gerarchicamente.

Il selettore dell'istruzione CASE può essere o un identificatore di tipo o un identificatore il cui tipo è stato già specificato: questo equivale a definire un campo il cui contenuto varierà in funzione del valore del tipo corrispondente.

Dal punto di vista semantico, ovviamente bisogna che tutti i nomi dei campi siano diversi fra loro, anche se compaiono in differenti campi varianti.

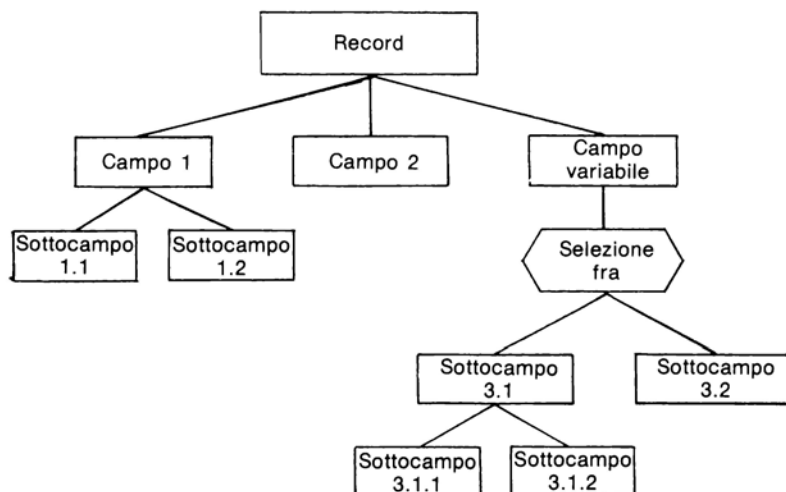
Un elenco di campi di questo tipo dev'essere organizzato in modo che i campi fissi precedano i campi varianti e che un elenco di campi possa contenere un solo elenco di campi varianti.

Un elenco di campi corrispondente ad una delle alternative può essere vuoto: in questo caso l'elenco vuoto è messo fra parentesi ( ).

Riferendoci all'esempio precedente, ora possiamo descrivere il record completo per mezzo delle seguenti dichiarazioni:

```
TYPE
   alfa: PACKED ARRAY[1 ... 10] OF char;
   materia = (letteratura, linguaviva, matematica, informatica);
   data = RECORD giorno: 1 ... 31;
                 mese: 1 ... 12;
                 anno: 0 ... 99
          END;
   voto: 0 ... 10;
   studente = RECORD
                 nome: RECORD cognome: alfa;
                              nome: alfa
                       END;
```

```
              data nascita: data;
              CASE controllo: materia OF
                 letteratura: (scritto: voto; orale: voto);
                 lingua:  (composizione: voto;
                          traduzione: voto; orale: voto);
                 matematica: (analisi, algebra, logica: voto);
                 informatica: (programmazione, hardware,
                              teoria: voto)
           END;
```

Come si può vedere, il record permette di definire delle strutture di dati sotto forma di alberi:

Quindi il linguaggio Pascal consente di usare strutture gerarchiche di dati, sulla scorta di linguaggi come il COBOL: anche in questo la suddivisione dei dati è strutturata ad albero. Il COBOL è per ora il linguaggio più comunemente usato nelle applicazioni gestionali.

1.3 - Come si usano i records in un programma

I records non possono essere citati globalmente in un'istruzione di assegnazione, perchè sono formati da elementi di tipi diversi.

È invece possibile riferirsi ai singoli elementi di un record: per far questo bisogna specificare il nome (identificatore) del record, seguito da un suffisso indicante l'identificatore di campo interessato.

I due identificatori sono divisi da un punto.

Esempi

1) Se il record *indirizzo* contiene gli elementi, o campi, definiti nell'esempio del paragrafo 1.1, ciascun campo sarà individuato da

```
indirizzo.nome
indirizzo.via
indirizzo.citta
indirizzo.codicepostale
```

2) Nel caso di una struttura di tipo vettore di records, bisogna precisare l'elemento interessato del vettore ed il campo:

```
Settebello[Bologna].ora: = 12
Settebello[Bologna].minuto: = 10
```

Il campo *ora* relativo all'elemento *citta* viene definito come uguale alle ore 12 per il valore *Bologna*.

Quindi una variabile di tipo record dev'essere sempre seguita da un identificatore che specifichi il campo interessato.

Questo metodo è relativamente oneroso, perchè impone di far riferimento a tutti i campi del record.

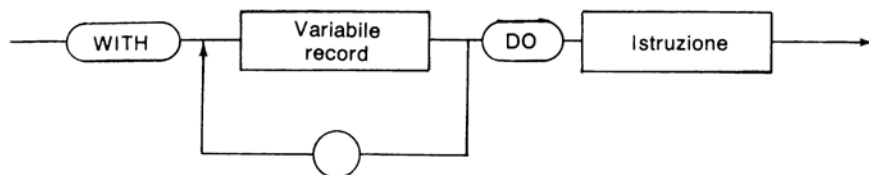
Nell'esempio del record *studente* presentato prima, se avessimo definito una variabile *s* di tipo *studente*, avremmo, ad esempio,

```
s.nome.cognome: = 'Rossi';
s.nome.nome: = 'Alberto';
s.data nascita.giorno: = 3;
s.data nascita.mese: = 6;
s.data nascita.anno: = 60;
s.controllo: = lingua;
s.composizione: = 6;
s.traduzione: = 7;
s.orale: = 8;
```

Esiste un'istruzione che permette di evitare la ripetizione dell'identificatore del record: è l'istruzione WITH, con la quale si può definire un blocco d'istruzioni che fanno riferimento a campi diversi del medesimo record.

1.4 - L'istruzione with (con)

Quest'istruzione permette di specificare i nomi dei records utilizzati in un blocco d'istruzioni; in tal modo si può fare a meno di far precedere tutte le variabili campo, corrispondenti a tali records, dalla notazione completa.

La sua sintassi è molto semplice:

Esempio

```
WITH indirizzo DO
BEGIN
  cognome: = 'Belli'; nome: = 'Aldo';
  via: = 'De Amicis';
  citta: = 'Milano';
  codicepostale.provincia: = 20;
  codicepostale.localita: = 123
END;
```

Tutto questo equivale all'insieme delle assegnazioni

```
indirizzo.cognome: = ...
indirizzo.nome: = ...
```

Si osservi tuttavia che un altro record, *codicepostale*, è sempre prefisso dei campi relativi; questo si sarebbe potuto evitare scrivendo

```
WITH indirizzo, codicepostale DO
BEGIN
  cognome: = 'Colonelli'; nome: = 'Carla';
  via: = 'Matteotti';
  citta: = 'Cremona';
  provincia: = 26;
  localita: = 100
END;
```

Qui i due records *indirizzo* e *codicepostale* sono specificati nell'argomento dell'istruzione WITH.

In generale, un annidamento d'istruzioni WITH (CON) come

```
WITH a1 DO
  WITH a2 DO
    .....
      WITH an DO istruzione;
```

che, in versione italiana, sarà

```
CON a1 FARE
  CON a2 FARE
    .....
      CON an FARE istruzione;
```

equivale alla forma

```
WITH a1, a2, ... an DO istruzione;
```

la cui versione italiana è

```
CON a1, a2, ... an FARE istruzione;
```

1.5 - Limitazioni

Nei parametri di un'istruzione WITH non possono esserci delle variabili il cui contenuto venga modificato dall'istruzione stessa.

Una struttura del tipo

```
WITH a DO
  BEGIN
    .....
    a: = a + b
```

non è ammessa.

Il nome di un dato corrispondente ad un campo di record è formato dal nome del record più il nome del campo: di conseguenza possiamo usare un nome d'identificatore di campo uguale ad un nome d'identificatore semplice senza far nascere errori di sintassi o ambiguità.

Ad esempio, nella seguente dichiarazione:

```
VAR
  numero: integer;
  indirizzo: RECORD
             nome: string;
             via: RECORD numero: integer;
                         nome: string
                  END;
             END;
```

gl'identificatori *numero* e *nome* compaiono due volte, ma a livelli diversi della struttura: il primo si riferisce la prima volta ad una variabile semplice intera (a cui nel programma si fa riferimento con *numero*), la seconda volta ad una variabile campo individuata con *indirizzo.via.numero*; a sua volta l'identificatore *nome*, che compare due volte nella struttura a record, la prima volta si riferisce alla variabile campo *indirizzo.nome*, la seconda volta alla variabile campo *indirizzo.via.nome*.

Non c'è nessuna ambiguità, ed è un tipo di dichiarazione ammesso in Pascal.

1.6 - Campi di tipo insieme

Possiamo definire records con campi di tipo *insieme*.

Esempio

Supponiamo di voler definire una distribuzione di carte da gioco:

```
TYPE
   colore = (fiori, quadri, cuori, picche)
   figura = (re, dama, fante)
   giococolore: RECORD
                      c: colore;
                      f: SET OF figura
                   END;
VAR
   distribuzione: giococolore;
```

Si potranno così avere assegnazioni del tipo

```
distribuzione.c = cuori;
distribuzione.f = [re, dama]
```

In tal modo possiamo definire un insieme di figure di un dato colore.

Non è possibile invece avere insiemi di records, perchè i records si definiscono soltanto per i tipi semplici.

Dal momento che, nonostante tutto, sono possibili i vettori di tipo record, possiamo definire

```
TYPE
   carta: RECORD
             c: colore;
             f: figura
           END;
   gioco: ARRAY[1 ... 13] OF carta;
```

Qui non c'è modo di evitare la presenza di più carte uguali nello stesso gioco.
Si è visto che possiamo anche definire

```
TYPE
  f: SET OF figura;
  gioco: ARRAY[colore]OF f;
VAR
  distribuzione: gioco;
```

con assegnazioni come

```
distribuzione[fiori]: = [dama, fante]
```

Si tratta cioè di un vettore di colori di carte formato da insiemi di figure.
Scrivendo invèce

```
TYPE
  gioco: ARRAY[1 ... 4] OF giococolore;
VAR
  distribuzione: gioco;
```

il risultato potrebbe essere questo:

```
i: = 2;
distribuzione[i].c: = picche;
distribuzione[i].c: = [fante, dama];
```

Assumendo che il vettore rappresenti i quattro colori, nella prima assegnazione si verificherebbe una certa ridondanza della variabile *i* e del campo *c*, perchè entrambi sarebbero associati al parametro *colore*; non si avrà invece ridondanza considerando *i* come uno dei quattro giocatori.

Vediamo quindi che i records permettono di definire strutture relativamente complesse, utilizzabili in tutti i casi in cui è necessario che appaia esplicitamente una struttura di dati particolare. Questa caratteristica sarà veramente preziosa con le strutture di tipo flusso, che vedremo più avanti.

2 - CONCETTI DI BASE SUI FLUSSI

2.1 - Il concetto di flusso

In informatica il concetto di flusso è un concetto generale, dal contenuto estremamente semplice. La sua definizione più generale è la seguente:

Un flusso è una serie d'informazioni binarie (bits) registrate su un supporto di memoria e definite per mezzo di un identificatore.

In parole povere, si tratta di una serie di dati numerici (o non numerici) registrati su un supporto di memoria che non è la memoria centrale. Parleremo quindi di flussi di schede e di nastri perforati, di flussi di stampa (anche la carta è un supporto di memoria), ma per lo più si usano supporti di tipo magnetico. Si può parlare anche di flussi "programma", perchè, come abbiamo appena visto, un programma può essere considerato come un flusso, quando è memorizzato su una memoria ausiliaria.

2.2 - I supporti dei flussi

Il concetto di supporto di memorizzazione è fondamentale per parlare di flusso in senso informatico.

Agl'inizi dell'informatica i supporti più comunemente usati erano le schede e/o i nastri perforati; oggi questi tipi di supporti, pur essendo ancora impiegati, vanno scomparendo, per la difficoltà di maneggiare ed immagazzinare flussi residenti su di essi. Inoltre il loro trattamento richiede tempi molto lunghi per la lentezza dei lettori di schede e nastri.

Abbiamo visto che i tabulati che si hanno in uscita da una stampante possono essere considerati come flussi.

Per quello che si è appena detto in questo capitolo, ci occuperemo soprattutto dei flussi su supporti magnetici, che hanno il grosso pregio di poter essere letti e scritti dal calcolatore ad una velocità relativamente molto alta.

Inoltre sono meno ingombranti, e più facili da maneggiare nelle elaborazioni informatiche; l'unico difetto è che per lavorare su di essi bisogna usare un calcolatore, sia per la lettura che per la scrittura.

Fra i supporti magnetici si possono distinguere: i supporti di tipo *sequenziale* (nastri e cassette magnetiche) ed i supporti di tipo *casuale*, come i dischi magnetici.

2.2.1 - I supporti di tipo sequenziale

I sistemi grossi e medi (compresi i minicalcolatori) dispongono di *nastri magnetici*, sui quali le informazioni sono registrate in forma numerica (binaria), e le modalità di codifica sono normalizzate secondo degli standards precisi. Il nastro magnetico permette di registrare fino a diversi milioni di caratteri, con una densità di registrazione che va, attualmente, da 800 a 9600 bpi (bit per inch); pertanto è l'ideale per memorizzare grossi flussi, in particolare archivi di dati.

Questo tipo di supporto non è disponibile sui microcalcolatori, perchè ha un costo proibitivo. Su di essi si trovano invece supporti sequenziali del tipo *cassetta magnetica*.

Esistono due tipi di dispositivi a cassetta: le *cassette digitali* e le *cassette audio*.

Sulle cassette digitali l'informazione è registrata secondo lo stesso principio dei nastri magnetici standard, per cui queste cassette sono affidabili quanto i nastri magnetici e si leggono abbastanza rapidamente. Per contro i registratori sono relativa-

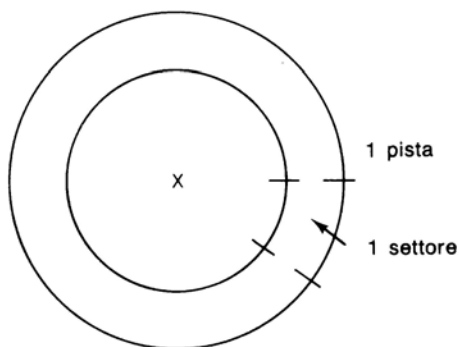
mente costosi. Il loro impiego è diminuito considerevolmente con la comparsa dei floppy disks e l'affermarsi delle cassette audio.

Di fatto, il supporto più economico e più diffuso sui piccoli microcalcolatori è la cassetta audio, che si può usare con i comuni registratori a cassette. Ma non esiste nessuno standard di codifica delle informazioni, e l'affidabilità varia a seconda dei sistemi. Inoltre la lettura dei dati è abbastanza lunga, a causa della diminuita velocità di lettura e scrittura.

Per piccoli flussi a cui si accede in forma sequenziale, in particolare nella memorizzazione di flussi programma, questa soluzione è del tutto soddisfacente.

2.2.2 - I supporti di tipo casuale

Un supporto è detto *casuale*, in contrapposizione ai supporti sequenziali, quando è possibile accedere all'informazione in modo diretto, o casuale, senza far scorrere l'intero supporto, dall'inizio alla fine, per leggere o scrivere un dato. Beninteso, questo non vuol dire che il supporto si comporta in maniera casuale, cioè probabilistica, perchè è sempre pilotato deterministicamente dal calcolatore: casuale è il tempo di accesso necessario per la lettura o la scrittura di un'informazione, tenuto conto delle letture o scritture effettuate prima. Si definisce pertanto un tempo di accesso medio.

Esistono dischi rigidi (hard disks) e dischi flessibili (floppy disks), entrambi costituiti da piste concentriche.

Una pista si divide in settori. Tutte le piste ed i settori sono individuati da un numero d'ordine, che prende il nome di indirizzo. Questo numero serve per specificare la lettura o la scrittura di una determinata pista o di un determinato settore.

2.2.2.1 - I dischi rigidi (hard disks)

Sono caratterizzati da un supporto rigido, ricoperto di materiale magnetico e chiuso in una cartuccia, o contenitore. Le testine di lettura e di scrittura non poggiano sul supporto magnetico: ne consegue una maggior velocità di trasferimento dei dati senza usura del supporto.

Fra i dischi rigidi, quelli *a testine fisse* hanno tante testine di lettura quante sono le piste. Il supporto è mobile, mentre le testine sono fisse, cosicchè si può accedere più rapidamente ad un qualunque settore. Il tempo di accesso è in media di 10 millisecondi. I dischi di questo tipo sono i più costosi, ed hanno lo svantaggio di non essere estraibili; di conseguenza non si possono usare per contenere archivi di dati, ma solo con sistemi per i quali il tempo di accesso è un parametro critico.

Ci sono poi i dischi *a testina mobile*, nei quali un'unica testina di lettura si sposta per posizionarsi sulla pista che interessa. Qui si ha un movimento del braccio che allunga il tempo di accesso, che può andare da 20 a 100 ms. (60 ms. in media), ma c'è il vantaggio che questi dischi sono meno costosi ed inoltre sono estraibili, per cui è possibile montare in fila diversi supporti. Con questi dischi poi si possono realizzare degli archivi: la loro capacità varia da diversi milioni a diverse decine di milioni di caratteri, nel caso di più supporti sovrapposti (*dispacks* in inglese).

Comunque si tratta in ogni caso di unità di registrazione relativamente costose, anche se gli sviluppi della tecnologia permettono di prevedere una considerevole riduzione del costo dei dischi non estraibili (i dischi di tipo *Winchester*).

I dischi rigidi sono necessari nell'elaborazione di volumi notevoli di dati, soprattutto nei sistemi di *data base*. Sui grossi e medi sistemi, compresi i minicalcolatori, questi dispositivi costituiscono quella che vien detta la "memoria di massa", che viene suddivisa fra diverse applicazioni e diversi utilizzatori.

2.2.2.2 - I dischi flessibili

Il principio fisico di registrazione e di accesso alle informazioni è lo stesso che per i dischi rigidi: la differenza sta nel materiale di cui è fatto il supporto (plastica ricoperta di uno strato magnetico) e nella maggior semplicità della meccanica dei registratori.

Anche qui le testine sono mobili, ma la testina di lettura poggia sul supporto. Il pregio di questi dischi sta nel basso costo e nell'ingombro limitato.

Ci sono diversi modelli, ma l'unico standard è attualmente lo standard IBM, relativo ai dischi da 250000 caratteri.

Recentemente sono apparsi anche i dischi doppia faccia, doppia densità, che permettono di quadruplicare la capacità degli altri modelli: in un solo disco si può memorizzare oltre un milione di caratteri (in un minidisco 500000 circa).

Comunque per i minidischi non esiste standard, per cui possono esserci problemi di lettura da parte di sistemi diversi.

Lo sviluppo di questa tecnologia ha reso possibile fornire di memorie di massa i microcalcolatori, rendendo possibile in particolare la realizzazione di applicazioni di tipo gestionale su microcalcolatori.

Questi supporti offrono oggi un'ottima affidabilità: a parte la degradazione dovuta all'usura (ma a lunghissimo termine), costituiscono senza alcun dubbio il sistema più economico per memorizzare flussi accessibili in modo diretto o casuale.

2.3 - I metodi di accesso ai flussi

Per metodo di accesso si deve intendere il modo con cui un flusso può essere trattato da un programma, indipendentemente dal supporto sul quale si trova.

2.3.1 - Flussi ad accesso sequenziale

In questi flussi le informazioni sono registrate in sequenza, l'una dopo l'altra: quindi per leggere un dato bisogna leggere tutti i dati precedenti, e analogamente, per scrivere o aggiungere un dato, bisogna posizionarsi alla fine del flusso. Questo tipo di accesso è il più semplice da realizzare e da trattare, ma è anche il meno flessibile. È adattissimo invece ai supporti sequenziali, come i nastri e le cassette magnetiche: è possibile trovare flussi ad accesso sequenziale su supporti che non lo sono!

Nel corso del capitolo vedremo degli esempi di trattamento di flussi sequenziali. Questi in genere sono organizzati in records successivi, di dimensioni che possono essere fisse o variabili a seconda dei programmi che lavorano su di essi.

2.3.2 - Flussi ad accesso diretto

Con questi flussi basta indicare il numero o la chiave che identifica il dato, oppure il record nel quale si trova il dato, per accedervi direttamente, indipendentemente dalla sua posizione fisica nel flusso. È evidente che, in pratica, questa tecnica si può applicare solo a supporti di tipo casuale; in alcuni casi essa stessa può esser vista come un metodo di accesso casuale, laddove è necessario passare per il tramite di una funzione di ripartizione casuale delle informazioni sul disco, essendo parametro di questa funzione la chiave relativa ai dati.

Questo metodo garantisce un accesso rapido ai dati del flusso.

2.3.3 - Flussi ad accesso indicizzato

Abbiamo qui un metodo di accesso diretto, o semidiretto, di tipo particolare: prima bisogna cercare, sequenzialmente, il valore della chiave in una tabella detta indice, per avere la posizione esatta dei dati o del record relativo.

Questo tipo di accesso ha il vantaggio di essere molto rapido, se gl'indici non sono troppo estesi, e se sono scanditi secondo un determinato ordine, pur lasciando intatta la possibilità di trattare il flusso in modo sequenziale.

Ad essi ricorreremo volendo trattare un flusso in modo sequenziale, ma mantenendo la possibilità di accedervi in modo rapido e pressochè diretto.

2.3.4 - Il metodo sequenziale indicizzato

Questo metodo è fondamentalmente analogo ai precedenti, ma si applica ad indici completamente definiti e gestiti dal sistema operativo del calcolatore. Allora l'indice corrisponde non al valore isolato di una chiave, ma ad un gruppo di chiavi, di cui l'indice specifica il primo e l'ultimo elemento.

Si cerca poi, in modo sequenziale, il record desiderato in un sottoinsieme del flusso. Questo metodo permette di considerare anche più livelli di indici. Di conseguenza la sua realizzazione è relativamente complessa, e di conseguenza conviene solo quando il numero delle possibili chiavi è molto alto. Questo metodo di accesso è assai usato nelle applicazioni gestionali, e va sempre più diffondendosi sui microcalcolatori orientati alle applicazioni gestionali.

2.3.5 - Organizzazione di un flusso

Quale che sia il metodo di accesso, in ogni caso i flussi sono generalmente organizzati in blocchi, o entità, che prendono il nome di *records logici*. Questo significa che la struttura del flusso è indipendente dalla struttura fisica dei settori sul supporto.

Il record logico è il solo ad avere significato dal punto di vista del programmatore. La dimensione dei records è fissa o variabile, a seconda delle applicazioni. Così la loro disposizione sul supporto può essere o no a blocchi: un record è bloccato quando ad un settore corrispondono diversi records, cosa che permette di sfruttare meglio lo spazio disponibile.

Quando può essere trattato sequenzialmente, un flusso termina sempre con un record particolare detto *fine del flusso* (*end of file* in inglese): la rilevazione della fine del flusso indica che l'elaborazione deve terminare.

2.4 - I sistemi operativi per dischi

In gergo informatico sono detti DOS (*Disk Operating System*). Sono formati da un certo numero di programmi, sviluppati dal costruttore, e fanno parte del software di sistema che viene fornito all'utilizzatore per la gestione dei flussi su disco. Pertanto un sistema operativo siffatto conterrà, almeno, la gestione delle etichette d'identificazione dei flussi e dello spazio disponibile sul disco.

Sui sistemi grandi e medi il DOS può essere molto complesso, e permette la gestione di flussi sequenziali diretti e sequenziali indicizzati.

Con i microcalcolatori le possibilità sono più ridotte, ma tendono rapidamente a crescere, soprattutto per quanto riguarda i sistemi rivolti alle applicazioni gestionali.

Pur non intendendo descrivere i vari tipi di DOS, val la pena di segnalare che, per sviluppare un'applicazione in cui occorre gestire dei flussi, è necessario disporre di un minimo di moduli operativi, detti *primitive del sistema di flussi*, ai quali il programmatore può accedere.

Le primitive di uso più comune sono: la *creazione* di un flusso, la sua *apertura*, la *lettura* e la *scrittura* di un record o di dati elementari, la *chiusura* di un flusso, la *cancellazione* di un flusso.

In generale il DOS comprende un certo numero di programmi di utilità: copia di un flusso, lista della direttrice del supporto, visualizzazione del contenuto di un flusso, etc. Beninteso, alcune operazioni possono venir programmate dall'utilizzatore, a patto che disponga delle funzioni elementari del sistema di gestione dei flussi.

Per il trattamento dei flussi in Pascal bisogna senz'altro poter usufruire di queste

primitive di sistema, che infatti sono rese disponibili per mezzo di specifiche istruzioni: comunque, prima d'illustrare come vengono trattati i flussi in Pascal, parleremo delle procedure standard d'ingresso/uscita.

L'ultima parte del capitolo sarà dedicata alle istruzioni suddette ed alla loro programmazione; semplici esempi serviranno a chiarire l'esposizione.

3 - LE PROCEDURE STANDARD D'INGRESSO/USCITA

In Pascal le uniche procedure standard sono le procedure d'ingresso e di uscita, oltre alle procedure di gestione dei flussi, di cui si parlerà più avanti in questo stesso capitolo.

Queste procedure sono già state abbondantemente utilizzate in tutti i programmi presentati finora, ma adesso ne parleremo in modo più approfondito.

In tutti i sistemi Pascal esistono due flussi standard assegnati ai dispositivi d'ingresso/uscita propri del sistema utilizzato: intendiamo parlare dei flussi *input* (*ingresso*) e *output* (*uscita*), che possono venir specificati come parametri del programma.

Esempio

```
PROGRAM esempio (input, output);
```

Questa frase indica, semplicemente, che il programma utilizza i dispositivi periferici assegnati per l'ingresso dei dati e l'uscita dei risultati.

Nella maggior parte dei sistemi quest'assegnazione è automatica: in particolare, nei microcalcolatori, il flusso *input* è riferito alla tastiera, il flusso di uscita (*output*) ad uno schermo catodico, cioè ad un video. Nei sistemi maggiori l'ingresso può avvenire da lettore di schede, mentre l'uscita può essere sulla stampante.

A questi due flussi corrispondono delle procedure d'ingresso (o di *lettura*) e delle procedure di uscita (o di *scrittura*).

3.1 - Le procedure di lettura

Sono due, e precisamente *read* (cioè *leggere*) e *readln* (cioè *leggere linea*), delle quali fin dall'inizio del libro abbiamo dato moltissimi esempi.

Tutti i dati vengono letti come una sequenza di caratteri: le conversioni opportune sono effettuate in funzione del tipo di variabile associato. Nel flusso *input* quindi ciascun dato può essere visto come una sequenza di caratteri.

L'elenco di parametri che compare in queste procedure non obbedisce alla sintassi ordinaria, perchè il numero dei parametri è variabile.

3.1.1 - La procedura read

L'elenco dei parametri corrisponde ad un elenco di variabili di programma, di tipo

carattere, intero o sottocampo (e, in alcuni casi, di tipo stringa di caratteri, quando questo tipo è considerato standard).

Il primo parametro, facoltativo, è un nome di flusso (v. l'esempio che segue); se è omesso, ci si riferisce implicitamente al flusso d'ingresso standard (*input*).

La forma generale è pertanto

```
read (f, v1, v2, ... vn)
```

ove *f* è un nome di flusso, mentre *v1*, *v2*, ... *vn* sono variabili appartenenti ai tipi detti prima. Nell'elenco possono comparire variabili di tipi diversi.

La forma *read* (*v1, v2, ... vn*) equivale alla forma

```
read (input, v1, v2, ... vn)
```

Quando il programma è eseguito, in fase d'introduzione dei dati i dati numerici, se ci sono, sono separati da caratteri di *spazio* (o di *blank*) o da caratteri di *fine linea* (ritorno carrello Ⓡ o avanzamento carta).

L'istruzione *read* (*f, v1,... vn*) è equivalente al blocco d'istruzioni

```
BEGIN
  read (f, v1);
  ....
  read (f, vn)
END;
```

3.1.2 - La procedura readln

È simile alla procedura *read* per quanto riguarda la funzione, che è quella di leggere dei dati, e la forma dell'elenco dei parametri.

Quindi abbiamo, in analogia alla procedura *read*, le forme

```
readln (f, v1, v2, ... vn)
```

o

```
readln (v1, v2, ... vn)
```

equivalente a

```
readln (input, v1, ... v2)
```

Differisce dalla procedura *read* per il fatto che, una volta letto l'ultimo valore *vn*, il resto della linea è ignorato.

Comunque con l'istruzione *readln* è possibile leggere i dati contenuti su più linee. Quindi la procedura *readln* (*f, v1, v2, ... vn*) è equivalente alla sequenza d'istruzioni

```
BEGIN
  read (f, v1);
  read (f, v2);
  .....
  read (f, vn);
  readln (f)
END;
```

3.2 - Le procedure di uscita

Anche per l'uscita esistono due procedure: *write* e *writeln*.

3.2.1 - La procedura write

È caratterizzata dalla presenza di un primo parametro facoltativo, che individua un flusso di uscita: se questo parametro non è specificato, l'uscita avverrà sul flusso *output*.

La forma generale di chiamata di questa procedura è

```
write (f, a1, a2, ... an)
```

ove *a1* ... *an* sono gli argomenti o parametri di uscita. La loro forma è

```
esp1: esp2: esp3
```

ma può essere anche

```
esp1: esp2
```

o semplicemente

```
esp1
```

Cioè un parametro di uscita è scomponibile in tre parti, costituite da espressioni separate dal carattere due punti (:).

La prima parte del parametro è un'*espressione* (*esp1*), il cui valore sarà scritto e decodificato a seconda del suo tipo: intero, reale, stringa di caratteri, o booleano. La seconda e la terza parte sono facoltative, e rappresentano delle espressioni che sono delle opzioni grazie alle quali si può specificare la forma dei risultati scritti.

La *seconda espressione* (*esp2*) garantisce il controllo della zona scritta, specificando il numero minimo di caratteri: quest'espressione dev'essere calcolata nella forma di un intero naturale. Se il numero dei caratteri da scrivere è minore di questo valore, prima della scrittura si avranno in uscita degli spazi bianchi; se invece supera il valore specificato, non si avrà troncamento.

Se quest'espressione manca, la scrittura avviene nel punto della linea in cui ci si trova.

La terza parte del parametro, data dalla *terza espressione* (*esp3*), interessa soltanto i dati di tipo reale, e indica il numero di cifre decimali che vengono dopo il punto decimale.

Se quest'espressione manca, il valore viene scritto in notazione mobile.

In conclusione, la procedura *write* (*f, a1, a2, ... an*) è equivalente all'istruzione composta

```
BEGIN
  write (f, a1);
  write (f, a2);
  .....
  write (f, an)
END;
```

3.2.2 - La procedura writeln

L'elenco degli argomenti, o parametri, della funzione è praticamente identico a quello della procedura *write*.

L'unica differenza sta nel fatto che, dopo che è stato scritto l'ultimo argomento *an*, si passa alla linea successiva.

Si ha quindi:

```
writeln(f, a1, a2, ... an)
```

che equivale all'istruzione composta

```
BEGIN
  write (f, a1);
  write (f, a2);
  .....
  write (f, an)
  writeln (f)
END;
```

4 - I FLUSSI IN PASCAL

Abbiamo visto che il linguaggio Pascal permette di definire delle strutture di dati relativamente complesse, dette records. Poi abbiamo introdotto alcuni concetti generali sui flussi, visti come insiemi di records o componenti.

Comunque si è visto che, in Pascal, il concetto di record non è necessariamente legato a quello di flusso.

Infatti il concetto di flusso (*file* in inglese) implica semplicemente una sequenza di componenti del medesimo tipo: quindi un flusso può essere formato altrettanto bene da sequenze di numeri, di caratteri o di records.

Quello che caratterizza un flusso è il fatto di esistere indipendentemente da un programma: prima e/o dopo la sua esecuzione.

In Pascal il concetto di flusso sussiste indipendentemente da un supporto fisico, ma presuppone che degli insiemi di informazioni, o di dati, siano memorizzati su un supporto fisico di memoria ausiliaria.

Il concetto è quindi collegato a quello, di tipo astratto, che interviene in Pascal a proposito di tutti i tipi di dati. Un flusso, comunque, può essere creato, e trattato, da programma.

Un altro punto che caratterizza un flusso è il fatto che la sua lunghezza non è definita una volta per tutte dal programma: esso infatti può essere aggiornato cancellando o aggiungendo degli elementi. Si tratta di una scrittura che evolve nel tempo, e pertanto lo stesso programma può lavorare più di una volta su uno stesso flusso.

In Pascal si assume che un flusso termini con uno speciale marcatore, detto *fine di flusso* (*eof*, dall'inglese End Of File).

In un dato istante il programma può lavorare su un solo elemento del flusso, il che presuppone l'esecuzione di una procedura iterativa da attivare fintantoché non si raggiunge l'elemento cercato (o la fine del flusso). Inoltre occorre avere in memoria una zona buffer che ospiti l'elemento in corso di elaborazione. Per riferirsi a quest'elemento esiste una variabile, detta a volte *variabile buffer*, dello stesso tipo degli elementi del flusso. Dato che compito di questa variabile è quello di far riferimento all'elemento corrente del flusso sequenziale relativo, possiamo paragonarla ad una *finestra*, spostabile lungo l'insieme degli elementi del flusso.

Esistono, d'altra parte, delle procedure standard con cui trattare gli elementi di un flusso: inizializzazione, riscrittura, avanzamento, aggiunta, lettura, scrittura.

4.1 - Dichiarazione dei flussi sequenziali in Pascal

In Pascal un flusso sequenziale viene dichiarato come un tipo di dato e può essere definito o nella descrizione di tipo di una variabile o in una dichiarazione di tipo.

Esempi

1)
```
VAR
    numero: FILE OF integer;
```

2) TYPE

```
TYPE
   coord = (x, y);
   tracciato = FILE OF coord;
VAR
   curva: tracciato;
```

Nel primo esempio l'identificatore del flusso è *numero*: si tratta di una sequenza di numeri interi, senza limiti di dimensioni (che ci sono invece per i vettori).

Nel secondo esempio l'identificatore è *curva*, che è di tipo *tracciato*, e rappresenta una sequenza di coordinate di punti (x, y): si vede quindi che non bisogna far confusione fra il tipo flusso ed il flusso medesimo.

La sintassi di una dichiarazione di flusso è data da un ramo del diagramma della dichiarazione di tipo:

Si possono definire dei tipi flusso formati da elementi di un qualsivoglia tipo di dati, compresi i tipi vettore e record.

Esempi

1) Avendo a che fare con un flusso di dati su schede perforate, è possibile definire un flusso di tipo vettore di caratteri in questo modo:

```
TYPE
   colonna = 1 ... 80;
   scheda = PACKED ARRAY[colonna] OF char;
VAR
   flusso: FILE OF scheda;
```

2) Volendo invece definire un flusso di indirizzi, dovremo assimilarlo ad un insieme di records Pascal.
Se definiamo un tipo record in questo modo:

```
TYPE
   stringa = PACKED ARRAY[1 ... 20] OF char;
   indir = RECORD
             nome: stringa;
             via: stringa;
             citta: stringa;
             provincia: integer;
             codice: integer
             END;
```

possiamo definire un flusso

indirizzo: FILE OF indir;

che sarà un flusso sequenziale di records di tipo *indir* (v. quanto si è detto prima).

Nelle applicazioni più usuali e, in particolare, in quelle gestionali, i flussi sono sempre strutturati in records, detti anche componenti: in Pascal, per evitare confusioni con il concetto di record, chiameremo *componente* l'elemento di un flusso. Un componente può ad esempio consistere in un numero, in un vettore, o in un record.

Nel caso di flussi memorizzati su memorie ausiliarie di tipo magnetico, spesso è preferibile definire strutture di tipo record che contengono più elementi per ciascun componente: in tal modo le operazioni d'ingresso/uscita sono ottimizzate meglio che se i componenti del flusso fossero formati da un solo elemento scalare.

Comunque, in determinate applicazioni, soprattutto scientifiche, non è escluso di poter pensare a flussi di vettori numerici non strutturati in records: in questo caso è vincente il concetto di flusso sequenziale, perchè offre tutta la flessibilità necessaria nelle applicazioni scientifiche e gestionali.

Inoltre sappiamo che in Pascal esistono due flussi standard sempre e comunque accessibili, anche se non sono stati dichiarati, e cioè il flusso d'ingresso (*input*) e quello di uscita (*output*). Questi sono assegnati alle periferiche d'ingresso/uscita standard del sistema su cui si lavora. Su questo torneremo fra poco.

4.2 - Il trattamento dei flussi sequenziali in Pascal

Un flusso è associato ad un identificatore di tipo variabile. Questa variabile tuttavia non può venire usata direttamente in un'istruzione di assegnazione, o in un'espressione, perchè è associata ad un insieme indeterminato di elementi.

Ad un flusso si può accedere soltanto elemento per elemento: a ciascun elemento è associato un tipo di base, definito con la dichiarazione di flusso (FILE OF).

Per accedere, o riferirsi, all'elemento corrente di un flusso, è quindi naturale servirsi dell'identificatore, seguito tuttavia da un marcatore, rappresentato da una freccia verticale.

Cosi, negli esempi precedenti, gli elementi dei flussi *numero*, *curva* e *indirizzo* sono individuati da *numero*↑, *curva*↑ e *indirizzo*↑: *numero*↑ rappresenta un elemento di tipo intero; *curva*↑ è associato ad un elemento di tipo coordinata; *indirizzo*↑ si riferisce ad un elemento di tipo record.

Queste variabili sono dette, nella terminologia Pascal, *variabili buffer*, o *variabili finestra*: noi, per le osservazioni precedentemente svolte, preferiamo chiamarle *variabili componente*. Infatti non bisogna confondere questo concetto con il concetto di puntatore, che vedremo più avanti. Formalmente, entrambi i tipi di variabile sono rappresentati da identificatori seguiti da un carattere freccia verticale, ma i loro significati sono diversi, e non si usano nello stesso modo.

Le variabili componente si possono usare in espressioni ed in istruzioni di assegnazione; il loro tipo è lo stesso di quello degli elementi o componenti del flusso.

Si osservi tuttavia che nella gestione di un flusso la *variabile componente* (o buffer), indicata con *f*↑ per un flusso il cui nome sia *f*, può essere completamente ignorata: in questo caso bisogna ricorrere alle procedure standard d'ingresso/uscita relative ai flussi. Il valore di *f*↑ poi può essere indeterminato, se il marcatore di fine flusso è raggiunto o superato, cioè se la funzione *eof* è vera.

4.3 - La funzione booleana eof (End Of File)

Questa funzione booleana è utilizzabile nel trattamento di un qualunque flusso: è vera se si raggiunge la fine di un flusso, falsa nel caso opposto.

Se la funzione *eof* non ha parametro, riguarda un flusso standard; se invece è associata ad un parametro flusso, ad esempio *eof* (*nomeflusso*), allora riguarda il flusso il cui nome è trasmesso come argomento della funzione.

4.4 - Le procedure per il trattamento dei flussi

Un flusso è identificato da un nome. Un componente, a sua volta, può essere individuato da una variabile associata al componente corrente del flusso sequenziale.

Le procedure standard, in Pascal, permettono di posizionare la variabile componente, o finestra, all'inizio del flusso, di cancellare o inizializzare un flusso, di leggere o aggiungere un componente.

4.4.1 - Posizionamento all'inizio del flusso

Quest'operazione è effettuata dalla procedura *reset* (*nomeflusso*): unico parametro, il nome del flusso. Oltre a ciò, questa procedura *legge il primo componente* del flusso e lo assegna alla variabile componente *nomeflusso* ↑ corrispondente.

Se il flusso è vuoto, la procedura rileva la fine del flusso, ed il valore di *eof* (*nomeflusso*) è vero, mentre il valore della variabile componente è indeterminato.

Se nel flusso c'è almeno un componente, il valore di *eof* (*nomeflusso*) è falso.

4.4.2 - Inizializzazione o creazione di un flusso

Il contenuto di un flusso è creato dalla scrittura entro il flusso stesso. Comunque, prima di scrivere per la prima volta in un flusso, o di riscriverlo completamente, bisogna attivare la procedura

```
rewrite (nomeflusso)
```

Quest'operazione ha lo scopo di posizionare la finestra all'inizio di un flusso vuoto e di prepararlo alla scrittura. Questo si fa tanto per *creare* un flusso quanto per *riscriverlo* completamente, la qual cosa presuppone la cancellazione e la distruzione della

versione preesistente: quindi la procedura agisce trasformando un flusso esistente in un flusso vuoto.

Inoltre, questa procedura ha l'effetto di posizionare *eof* (*nomeflusso*) al valore vero.

4.4.3 - Scrittura di un componente del flusso

Quest'operazione è effettuata dalla procedura *put* (cioè *mettere*).

Il parametro è sempre il nome del flusso:

```
put (nomeflusso)
```

La procedura fa sì che il contenuto della variabile componente sia aggiunto alla fine del flusso indicato dal marcatore associato a quella variabile. Quindi la procedura sortisce il suo effetto solo se il predicato booleano *eof* (*nomeflusso*) è vero: in questo caso viene scritto il componente, la finestra è avanzata ed il predicato *eof* rimane vero.

4.4.4 - Lettura di un componente del flusso

È l'operazione inversa della precedente, e presuppone che sul flusso sia già stata effettuata una scrittura.

La procedura che effettua l'operazione è *get* (cioè *prendere*), e il parametro è anche qui il nome del flusso:

```
get (nomeflusso)
```

L'effetto è l'avanzamento della finestra corrispondente alla variabile componente, e l'assegnazione a quella variabile del contenuto del componente letto.

Se il predicato *eof* è vero, non si ha lettura, e il valore della variabile componente è indeterminato; se invece il predicato *eof* è falso, il componente viene letto.

4.5 - Il trattamento dei flussi su disco

Sappiamo che questo trattamento richiede un supporto logico di tipo DOS, che chiameremo sistema di gestione dei flussi su disco (SGFD).

All'SGFD si può accedere, da un lato, per mezzo dei comandi di sistema (comandi di monitor), dall'altro, da programma, per mezzo delle primitive di gestione dei flussi: così, ad esempio, per accedere alla lista dei flussi su disco, ci serviremo di un comando che richieda tale lista, mentre per scrivere un flusso di dati useremo le procedure Pascal standard.

Considereremo ora il caso di flussi su floppy disk, con esempi riferiti ad un sistema Apple U.C.S.D. con doppia unità a dischi: resta inteso che comandi analoghi devono essere presenti su qualunque sistema che disponga di un DOS.

4.5.1 - Formattazione di un dischetto

Sui dischetti vergini, così come arrivano dal fornitore, bisogna prima di tutto fare un'operazione detta di formattazione o inizializzazione del supporto. Ogni sistema prevede una tecnica particolare per organizzare i dati sul dischetto, per cui quest'operazione è necessaria anche se, ad esempio, si usa lo standard IBM.

Per provare programmi di gestione dei flussi, è opportuno usare un dischetto vergine, ma formattato ed inizializzato.

Un SGFD è caratterizzato tipicamente dal modo con cui è gestito lo spazio libero di dischetto e da come sono aggiornate le etichette di flusso, essendo ciascun flusso caratterizzato da un *nome esterno*, registrato nella direttrice del supporto.

4.5.2 - Il trattamento dei flussi sequenziali su disco

Consideriamo ora i flussi di dati come sono trattati da programmi in Pascal. Queste elaborazioni fanno capo all'SGFD (DOS), per il tramite di primitive o d'istruzioni di SGFD, che possono essere richieste dal programma in questione. Qualunque sia il sistema utilizzato, le loro funzioni sono sempre le medesime:
- apertura di un flusso (*reset* o *rewrite*);
- chiusura (*close*);
- lettura (*read* o *get*);
- scrittura (*write* o *put*).

4.5.3 - Legame fra il nome interno ed il nome esterno di un flusso

Quando si ha un dato flusso su supporto magnetico, designato con un *nome esterno*, bisogna stabilire un legame fra questo nome e quello usato a livello di programma, cioè il nome interno. Se si lavora su grossi sistemi, si ricorrerà al linguaggio di controllo del sistema di gestione dei flussi, per fare le opportune assegnazioni, ma in modo esterno al programma.

Con i piccoli sistemi su microcalcolatore invece l'assegnazione può avere luogo direttamente all'interno del programma, associando, in fase di apertura del flusso, il suo nome interno (primo parametro) al suo nome esterno (secondo parametro), cosi da costituire una stringa di caratteri.

I sistemi Pascal U.C.S.D. dispongono delle seguenti procedure:

```
reset (nomeflussointerno, nomeflussoesterno)
```

e

```
rewrite (nomeflussointerno, nomeflussoesterno)
```

Nomeflussointerno dev'essere una variabile di tipo file; *nomeflussoesterno* una variabile di tipo stringa di caratteri.

Il secondo parametro può poi anche prendere direttamente la forma di una stringa di caratteri, posta fra apici.

Esempio

Si abbia un flusso dal nome interno definito dalla dichiarazione seguente:

```
VAR
  flus: FILE OF integer;
```

Volendo memorizzare questo flusso su un flusso esterno, denominato *numeri*, possiamo ricorrere alle seguenti soluzioni:

– in scrittura (creazione):

```
rewrite (flus, 'numeri')
```

– oppure, in lettura:

```
reset (flus, 'numeri')
```

Possiamo anche definire la variabile

```
VAR
  nomesterno: string;
```

e in tal caso adottare la seguente soluzione:

```
nomesterno: = 'numeri'
```

e le procedure

```
rewrite (flus, nomesterno)
```

oppure

```
reset (flus, nomesterno)
```

Così facendo, le procedure *get* e *put* non devono più stabilire il legame di cui sopra: esse necessitano solo del parametro *nomeflussointerno*.

4.5.4 - Chiusura di un flusso

Nella scrittura su disco o su nastro magnetico, la scrittura dell'ultimo componente del flusso non include quella del marcatore di fine flusso (*eof*), nè l'aggiornamento delle tabelle del sistema di gestione dei flussi, operazioni che sono necessarie per poter usare, e conservare dopo l'esecuzione del programma, i flussi destinati a non essere usati una sola volta.

Sul sistema U.C.S.D. la chiusura di un flusso è effettuata tramite la procedura *close* (cioè *chiudere*), e prevede due parametri, il nome del flusso ed un parametro opzionale:

```
close (nomeflusso, opzione)
```

Dopo l'esecuzione di questa procedura, la variabile componente è indefinita.

Le opzioni disponibili sono:

- l'opzione *normal*, che determina semplicemente la chiusura del flusso. Se questo è un flusso aperto in scrittura con la procedura *rewrite*, viene cancellato dalla direttrice, dato che corrisponde ad un flusso temporaneo.
- L'opzione *lock* (cioè *bloccare*), che permette di conservare i flussi e di aggiornare la direttrice.
- L'opzione *purge* (cioè *epurare*), che permette di chiudere il flusso, ma ne distrugge l'identificatore nella direttrice.
- L'opzione *crunch*, che permette di congelare il flusso in corrispondenza dell'ultimo accesso in lettura o in scrittura. Quindi il flusso verrà chiuso nel punto in cui ha avuto luogo l'ultima richiesta di una procedura *put* o *get*.

4.6 - Un'applicazione: un flusso d'indirizzi

Supponiamo di voler creare e trattare un flusso d'indirizzi descritti da un record.

Il programma che segue permette d'introdurre da tastiera degl'indirizzi, costituiti da cognome, nome e indirizzo; tutto ciò viene scritto in un flusso d'indirizzi.

4.6.1 - Creazione del flusso

```
PROGRAM flussoindirizzo;
TYPE
  indirizzo = RECORD
              cognome: string; nome: string;
              via: string; no: integer;
              citta: string;
              cap: 0 ... 99999
              END;
  flus = FILE OF indirizzo;
VAR
  ind: flus;
  componente: indirizzo;
(*programma principale*)
BEGIN
  rewrite (ind, 'indirizzo.text');
  WITH componente DO
    BEGIN
      WHILE cognome < > 'no' DO
```

```
          BEGIN
            write ('cognome:'); readln (cognome);
            write ('nome:'); readln (nome);
            write ('via:'); readln (via);
            write ('no:'); readln (no);
            write ('citta'':'); write (citta);
            write ('cap:'); write (cap);
            ind↑: = componente;
            put (ind);
            writeln ('altro indirizzo?');
            readln (cognome);
          END;
      END;
      close (ind, lock);
  END.
```

Qui il nome interno del flusso è *ind*, quello esterno *indirizzo.text*.

Esecuzione

```
cognome: Tripoli
nome: Ornella
via: Caracciolo
no: 83
città: Napoli
cap: 80136

altro indirizzo? s  Ⓡ

cognome: Borioni
nome: Michele
via: LIvorno
no: 7
città: Roma
cap: 00162

altro indirizzo? s  Ⓡ

cognome: Lojodice
nome: Tina
via: Congedo
no: 9
città: Lecce
cap: 73100

altro indirizzo? no  Ⓡ
```

4.6.2 - Rilettura del flusso

Il programma che segue permette di rileggere il flusso che abbiamo or ora considerato, e conservato sul dischetto grazie all'operazione di chiusura *close*.

```
PROGRAM flussoindirizzo;
TYPE
  indirizzo = RECORD
              cognome: string; nome: string;
              via: string; no: integer;
              citta: string;
              cap: 0 ... 99999
              END;
  flus = FILE OF indirizzo;
VAR
  ind: flus;
  componente: indirizzo;
  n: integer;
(*programma principale*)
BEGIN
  reset (ind, 'indirizzo.text');
  WITH componente DO
    BEGIN
      n: = 1;
    WHILE NOT eof (ind) DO
      BEGIN
        writeln;
        writeln ('indirizzo no', n);
        writeln;
        componente: = ind↑;
        write ('cognome:'); write (cognome: 15);
        write ('nome:'); writeln (nome);
        write ('via:'); write (via);
        write ('no:'); writeln (no: 5);
        write ('citta'':'); writeln (citta);
        write ('cap:'); writeln (cap);
        get (ind);
        n: = n + 1;
      END;
    END;
END.
```

Si osservi che la chiamata della procedura *get* ha luogo al termine del ciclo WHILE, perchè la procedura *reset* legge il primo componente nella variabile *ind↑*.

La stampa del flusso d'indirizzi quale è realizzata dal programma è:

```
indirizzo no 1
cognome: Tripoli      nome: Ornella
via Caracciolo no: 5
città: Napoli
cap: 80136

indirizzo no 2
cognome: Borioni      nome: Michele
via Livorno no: 7
città: Roma
cap: 00162

indirizzo no 3
cognome: Lojodice     nome: Tina
via Congedo no: 9
città: Lecce
cap: 73100
```

4.6.3 - Aggiornamento del flusso

Quest'operazione presuppone che il flusso venga letto fino ad incontrare il marcatore di fine flusso; poi si usa la procedura standard *put*: il componente è scritto *alla fine* del flusso. Il programma che si ottiene è il seguente:

```
PROGRAM flussoindirizzo;
TYPE
  indirizzo = RECORD
              cognome: string; nome: string;
              via: string; no: integer;
              citta: string;
              cap: 0 ... 99999;
              END;
  flus = FILE OF indirizzo;
VAR
  ind: flus;
  componente: indirizzo;
  n: integer;
(*programma principale*)
BEGIN
  reset (ind, 'indirizzo.text');
  WITH componente DO
    BEGIN
      n: = 1;
    WHILE NOT eof(ind) DO
```

```
            BEGIN
              get (ind);
              n: = n + 1;
            END;
              writeln;
              writeln ('indirizzo no',n);
              writeln;
              write ('cognome:'); readln (cognome);
              write ('nome:'); readln (nome);
              write ('via:'); readln (via);
              write ('no:'); readln (no);
              write ('citta":'); readln (citta);
              write ('cap:'); readln (cap);
              ind↑: = componente;
              put (ind);
              n: = n + 1;
          END;
      END.
```

4.7 - Uso delle procedure standard di lettura e di scrittura

Le procedure che abbiamo esaminato sono sufficienti per trattare qualsiasi problema. Comunque il contenuto di un componente può esser letto in una variabile diversa dalla variabile componente associata alla finestra corrente del flusso, ed analogamente è possibile scrivere un componente a partire da una variabile diversa dalla variabile componente associata al flusso.

Per far questo, bisogna usare la procedura standard

```
read (nomeflusso, variabile)
```

che è equivalente alla sequenza d'istruzioni

```
variabile: = nomeflusso↑;
get (nomeflusso);
```

Qui la *variabile* riceve il contenuto del componente letto, posto nella variabile componente *nomeflusso↑*.

Analogamente useremo in scrittura la procedura

```
write (nomeflusso, variabile)
```

che è equivalente a

```
nomeflusso↑: = variabile;
put (nomeflusso);
```

In questo caso la *variabile* contiene il valore del componente, che è quindi trasmesso alla variabile componente *nomeflusso↑*.

Come esercizio, si suggerisce di modificare i programmi che precedono usando le procedure standard *read* e *write*.

4.8 - Flussi di testo e flussi standard

In Pascal i flussi possono essere interni ad un programma o ad una procedura, o possono essere esterni, memorizzati su memorie ausiliarie.

Quando un flusso è definito in forma di sequenza di caratteri, parliamo di *flusso di testo*.

Le modalità di scrittura di un flusso-testo non sono diverse da quelle degli altri flussi: la variabile finestra (o componente) può essere di tipo carattere se il flusso è scritto carattere per carattere.

Il problema che si pone in questo caso è quello di suddividere il flusso in righe ed in pagine di testo: quest'operazione è talora indicata per mezzo di un carattere che significa *andare a capo*. Ora, in Pascal, il tipo carattere comprende soltanto i caratteri stampabili, quindi, per indicare l'andare a capo, bisogna ricorrere ad un meccanismo appropriato: abbiamo già visto che questo si può ottenere con le procedure *readln* e *writeln*.

4.8.1 - La funzione booleana eoln (End Of Line)

Questa funzione booleana è utilizzabile nel trattamento dei flussi - testo.

È vera quando si raggiunge un carattere *separatore di linea*, falsa nel caso opposto.

Se la funzione *eoln* non ha parametro, allora riguarda il flusso standard *input* (cioè *ingresso*). Se invece è associata ad un parametro flusso, come ad esempio in

```
eoln (nomeflusso)
```

siamo in presenza di un flusso - testo il cui nome è definito come parametro.

4.8.2 - I flussi standard

Sappiamo che esistono due flussi standard d'ingresso/uscita (*input*, *output*): questi sono sempre di tipo testo, e corrispondono alle dichiarazioni implicite

```
TYPE
  testo = FILE OF char;
VAR
  input, output: testo;
```

In alcune realizzazioni il tipo *testo* è standard, e corrisponde ad un vettore di caratteri impaccati.

Quando si usano le procedure d'ingresso/uscita standard, il parametro relativo al nome del flusso è facoltativo; tuttavia, in Pascal standard, dev'essere specificato come parametro del programma. Quando con i flussi standard si usano le procedure *read* e *write* nel leggere dei dati numerici (interi o reali), nel momento dell'introduzione avviene una conversione automatica dal modo carattere al modo di rappresentazione interna dei numeri; in uscita si ha la conversione inversa.

4.8.3 - Lettura e scrittura dei flussi di testo

Lettura e scrittura avvengono carattere per carattere. Chiaramente, come con tutti i flussi, si possono usare le procedure *put* e *get*. Allora, se il flusso si chiama *flusso* e il carattere in questione *car*, dopo l'apertura del flusso avremo:

— in scrittura

```
flusso↑: = car;
put (flusso);
```

— in lettura

```
car: = flusso↑;
get (flusso);
```

Usando le procedure standard d'ingresso/uscita, avremmo:

```
write (flusso, car)
```

e

```
read (flusso, car)
```

Con i flussi standard (*input*, *output*), queste procedure si riducono alla forma

```
write (car);
read (car);
```

Questo modo di usare le procedure standard d'ingresso/uscita è senz'altro molto pratico quando si verifica un programma, ma è pericoloso quando si vuol fare un programma che non solo individui un errore nel tipo dei dati introdotti, ma inoltre permetta all'utilizzatore che lavora in maniera interattiva di correggere l'errore in questione. Su questo ritorneremo in seguito, ma diciamo subito che, per avere programmi standard in cui sia possibile correggere un dato errato, bisogna lavorare con flussi-testo ed usare proprie procedure di conversione. Nei programmi presentati nel libro non abbiamo seguito questa indicazione perchè quello che soprattutto c'interessava era-

no gli algoritmi e le strutture fornite dal linguaggio per la programmazione. Comunque queste procedure saranno indispensabili se si vogliono ottenere programmi operativi in un contesto produttivo.

Ciò detto, precisiamo che l'impiego di flussi non standard il cui tipo sia dichiarato nel programma non pone particolari problemi, dal momento che la rilettura utilizza gli stessi tipi usati nella scrittura.

4.8.4 - Le procedure di fine linea

a) *In scrittura*

La procedura

writeln (flusso)

oppure

writeln se *flusso* = *output*

fa terminare la linea in uscita.

Esempio

Supponiamo di voler stampare un flusso-testo su una stampante a venti colonne, e che il flusso corrispondente si chiami *stampa*. Il programma sarà il seguente:

```
PROGRAM ingressouscita;
CONST
  punto = '.';
  lg = 20;
VAR
  stampa: text;
  car: char;
  n: integer;
BEGIN
  rewrite (stampa, 'printer:');
  REPEAT
  n: = 0;
    REPEAT
      read (car);
      write (stampa, car);
      n: = n + 1;
    UNTIL (n = lg) OR (car = punto);
  writeln (stampa);
  UNTIL car = punto;
END.
```

Il flusso *stampa* è dichiarato di tipo *text*, che è un tipo generalmente riconosciuto dai sistemi Pascal standard.

Il flusso viene aperto in scrittura (*rewrite*); è effettuata poi una lettura carattere per carattere, seguita da una scrittura e da una fine linea dopo 20 caratteri, a meno che non si trovi il carattere punto con cui è indicata la fine della stampa.

La procedura *rewrite* sarebbe stata inutile con un flusso standard.

Nel programma, verificato su un sistema Pascal U.C.S.D., il nome esterno del flusso corrispondente alla stampante prende il nome di '*printer*': con la periferica '*console*' come flusso di uscita, avremmo visualizzato esattamente tutte le operazioni d'ingresso/uscita.

b) *In lettura*

Per leggere un flusso-testo fino alla fine e scriverlo su una stampante rispettando i separatori di linea introdotti, ci serviremo del programma seguente:

```
PROGRAM ingressouscita;
VAR
  stampa: text;
  tastiera: text;
  car: char;
  n: integer;
BEGIN
  reset (tastiera, 'console:');
  rewrite (stampa, 'remout:');
  WHILE NOT eof(tastiera) DO
    BEGIN
      WHILE NOT eoln(tastiera) DO
        BEGIN
          read (tastiera, car);
          write (stampa, car);
        END;
  writeln (stampa);
  readln (tastiera);
END.
```

Qui la console viene usata come flusso d'ingresso, la stampante come flusso di uscita, denominato '*remout:*'.

Esecuzione

```
1 2 3 4 5 6 7 8 9 0 (R)                              tastiera
a b c d e f g h i j k l m n o p q x y z (R)          tastiera
1 2 3 4 5 6 7 8 9 0                                  stampante
a b ..  z                                            stampante
```

Quindi questo programma riproduce le righe del testo così come sono state introdotte.

4.8.5 - I flussi interattivi

Riprendiamo il programma precedente, usando il flusso *console* al posto del flusso *remout*: il flusso *console* è di volta in volta un flusso d'ingresso e di uscita, a seconda che si consideri la tastiera o lo schermo di visualizzazione.
Allora, eseguendo il medesimo programma, avremo i seguenti risultati:

```
1 2 1 3 2 4 3 5 4 6 5 7 6 8 7 9 8 0 (R) 9
0
a b a c b d c e d f e g f h g i h j i k j l k m
l n m o n p o q (R) p
```

Qui possiamo vedere un "tranello" delle procedure standard *reset* e *read*: l'uscita presenta un "ritardo" di un carattere rispetto all'ingresso, in quanto la procedura *reset* realizza la lettura del primo componente del flusso d'ingresso, e quindi bisogna attendere l'introduzione di un altro carattere perchè esca il precedente. Infatti l'istruzione

```
read (flusso, car)
```

è equivalente a

```
car: = flusso↑
get (flusso)
```

Tutto ciò può essere fastidioso nelle applicazioni interattive. Il Pascal U.C.S.D. prevede la modalità di *flusso interattivo:* modifichiamo in tal senso il programma precedente, sostituendo semplicemente il tipo *text* del flusso tastiera con il tipo *interactive*:

```
PROGRAM ingressouscita;
VAR
   stampa: text;
   tastiera: interactive;
   car: char;
   n: integer;
BEGIN
   reset (tastiera, 'console:');
   rewrite (stampa, 'console:');
   WHILE NOT eof(tastiera) DO
```

```
        BEGIN
          WHILE NOT eoln(tastiera) DO
            BEGIN
              read (tastiera, car);
              write (stampa, car);
            END;
          writeln (stampa);
          readln (tastiera);
        END;
    END.
```

All'esecuzione si ha:

```
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 0 0(R)
a a b b c c d d e e f f g g h h i i j j k k(R)
```

Questa volta l'introduzione del carattere è seguita immediatamente dall'uscita di questo stesso carattere.

Quindi un flusso di tipo *interactive* è un flusso di tipo *testo* per il quale la procedura *read* (*flusso, car*) è l'inverso della procedura standard: abbiamo cioè l'equivalente delle due operazioni successive

```
get (flusso)
```

e

```
car: = flusso↑
```

Analogamente, la procedura *reset* posiziona la variabile componente all'inizio del flusso, ma non ne legge il primo componente: per avere lo stesso effetto del *reset* standard, bisogna far seguire un'operazione di *get*.

AVVERTENZE

1) È utilissimo analizzare bene questi brevi programmi per capire il funzionamento delle procedure standard che lavorano sui flussi - testo.

2) In alcuni sistemi, quando il flusso è inviato su una stampante, il primo carattere di ogni linea è usato come carattere di controllo: questi caratteri saranno generalmente:
   - spazio per l'andata a capo (interlinea);
   - 1 per saltare ad inizio pagina;
   - + per stampare sulla stessa linea (sovrascrittura);
   - 0 per saltare una linea.

4.8.6 - La procedura di cambio pagina

La maggior parte dei sistemi Pascal prevede una procedura che permette di cambiar pagina in un flusso - testo o interattivo.

Questa procedura si realizza inviando il carattere di "*form feed*", che fa avanzare un flusso stampante ad inizio pagina.

La chiamata della procedura è la seguente:

```
page (nomeflusso);
```

In particolare, per posizionarsi all'inizio del flusso di uscita standard output, si scriverà

```
page (output);
```

Se il flusso di uscita è la console di visualizzazione, si avrà la pulizia dello schermo e si comincerà a scrivere dall'alto a sinistra.

Esempio

Visualizzazione sullo schermo di un menù:

```
PROGRAM menuselezione;
VAR
  n, i: integer;
  s: ARRAY[1 ... 12] OF string;
  titolo: string;
PROCEDURE visualizzamenu;
VAR
  j: integer;
BEGIN
  page (output);
  writeln (titolo: 15);
  FOR j: = 1 TO n DO
    BEGIN
      writeln;
      writeln (j: 2,' ',s[j]);
    END;
END;
BEGIN
  WHILE NOT eof DO
    BEGIN
      read (titolo);
      readln (n);
      FOR i: = 1 TO n DO readln (s[i]);
      visualizzamenu;
    END;
END.
```

4.9 - I flussi ad accesso diretto

Il Pascal standard consente di definire soltanto flussi sequenziali.

Tuttavia, quando il flusso è un insieme di componenti strutturati a records di dimensione calcolabile sulla base della relativa dichiarazione, è possibile calcolare la posizione esatta di ciascun record: in tal modo si può accedere ad un componente in maniera diretta, ossia casuale, pur essendo il flusso fisicamente costituito da un flusso sequenziale.

Di fatto il Pascal U.C.S.D. permette di accedere direttamente al record che si desidera, per mezzo di una procedura di ricerca (*seek*). Con riferimento a questa versione, modifichiamo il programma sul flusso d'indirizzi presentato nel paragrafo 4.6, specificando la lunghezza massima di tutte le stringhe di caratteri definite nel programma in questione. Avremo pertanto:

```
PROGRAM flussoindirizzo;
TYPE
  indirizzo = RECORD
               cognome: string[20]; nome: string[10];
               via: string[30]; no: integer;
               citta: string[20];
               cap: 0... 99999
               END;
  flus = FILE OF indirizzo;
VAR
  ind: flus;
  componente: indirizzo;
(*programma principale*)
BEGIN
  rewrite (ind, 'indirizzo.dato');
  WITH componente DO
    BEGIN
      WHILE cognome <> 'no' DO
        BEGIN
          write ('cognome:'); readln (cognome);
          write ('nome:'); readln (nome);
          write ('via:'); readln (via);
          write ('no:'); readln (no);
          write ('citta'':'); readln (citta);
          write ('cap:'); readln (cap);
          ind↑: = componente;
          put (ind);
          writeln ('altro indirizzo?');
          readln (cognome);
        END;
```

```
        END;
        close (ind, lock);
    END.
```

Questo programma permette di scrivere dei records componenti la cui dimensione è calcolabile: è allora possibile attivare la procedura di ricerca in accesso diretto di un componente.

4.9.1 - La procedura di accesso diretto (seek)

Questa procedura ha due parametri: il nome del flusso ed il numero del record. La sequenza di richiesta è

```
seek (nomeflusso, nocomponente)
```

Il suo effetto è di posizionare il puntatore componente sul record componente desiderato: per ottenere il record corrispondente basta richiedere una procedura *get*.

4.9.2 - Lettura in accesso diretto di un componente

Bisogna sapere che il primo record del flusso ha numero d'ordine zero, e che non si possono fare due richieste della procedura *seek* senza intercalare fra l'una e l'altra un'operazione d'ingresso/uscita sul flusso (*get* o *put*).

Quindi, relativamente all'esempio del flusso d'indirizzi, il programma di lettura viene così modificato per realizzare l'accesso diretto: è richiesto il numero di un indirizzo, che diventa il secondo parametro della procedura di ricerca (*seek*); poi si attiva la procedura di stampa, che abbiamo già usato per l'accesso sequenziale.

4.9.3 - Aggiunta e modifica in accesso diretto di un componente in un flusso

La procedura di scrittura a fine flusso (*appendere*) è la stessa di un flusso sequenziale: prima di tutto bisogna posizionarsi alla fine del flusso, secondo la procedura che abbiamo già visto, poi si scrive il componente.

È possibile realizzare in accesso diretto la *modifica* di un componente: basta posizionarsi sulla posizione del componente da modificare ed eseguire la procedura *put*.

Concretamente, in modalità interattiva, è meglio verificare prima che il componente sia proprio quello che interessa, e poi modificarlo. Per far questo si dovranno eseguire le seguenti operazioni:

- seek (flus, nocomponente)
- get (flus)
- visualizzazione (scrittura) componente
- correzione componente
- seek (flus, nocomponente)
- put (flus)

Di fatto bisogna riposizionarsi sul componente, perchè il primo *get* fa avanzare la variabile componente di una posizione.

4.9.4 - I flussi indicizzati

Un flusso indicizzato è costituito da componenti scritti in sequenza sul disco, ai quali però si può accedere per mezzo di un indice, diverso dal puro e semplice numero di componente.

Un indice è una tabella, che viene esplorata sequenzialmente, e che indica la collocazione di ciascun componente entro il flusso: può essere schematizzato con una sequenza di coppie: *nome componente*, *indirizzo componente*.

Il nome del componente può anche essere un numero, ed il suo indirizzo è dato dalla posizione, rispetto all'inizio del flusso, espressa in numero di caratteri.

Se l'indice è di lunghezza fissa, sarà posto all'inizio del flusso; se invece è di lunghezza variabile, sarà posto alla fine. Quest'ultima soluzione è preferibile perchè più generale, ma richiede un'elevata sicurezza di funzionamento: infatti, se non è possibile riscrivere l'indice, o se è stato riscritto male, il flusso è difficilmente recuperabile.

La struttura di un flusso indicizzato è schematizzabile in questo modo:

Indirizzo indice	
1° componente	
2° componente	
...	
n-esimo componente	
n° comp. indirizzo	
n° comp. indirizzo	Indice
...	
n° comp. indirizzo	

Se l'indice è formato dai nomi dei componenti, questi non devono essere scritti nell'ordine 1°, 2°, etc., bensì nell'ordine richiesto.

4.10 - Conclusioni

In questo capitolo abbiamo esposto i concetti fondamentali per il trattamento dei dati strutturati in flussi. Si sono presentati programmi semplici, ma tuttavia rappresentativi dei vari modi di elaborazione disponibili su tutti i sistemi. Le primitive di SGFD sono analoghe su tutti i sistemi, anche i più grossi: invece il sistema di gestio-

ne dei flussi dev'essere sofisticato, se si vuole tener conto della suddivisione dei flussi fra più utilizzatori e dello sviluppo di sistemi di base di dati. Ad ogni modo quanto è attualmente disponibile sui sistemi piccoli, quelli a floppy disk, è sufficientemente sofisticato da consentire agli utilizzatori di sviluppare sistemi di memorizzazione dei dati strutturati in forma di piccole basi di dati (*data base*): in tal modo non si pone il problema della suddivisione contemporanea dei flussi o della base di dati. Lo sviluppo dell'approccio dei "sistemi distribuiti" permetterà in un futuro prossimo di rendere accessibili i dati memorizzati sui sistemi più piccoli da parte di un qualunque terminale, o sistema, collegato ad una rete.

ESERCIZI

1. Scrivere le procedure di calcolo relative ai numeri complessi, usando il tipo record

```
TYPE
complesso = RECORD partereale: real;
                   parteimmag: real
            END;
```

Richiamiamo le seguenti formule di calcolo:

Addizione

$$(a + ib) + (c + id) = (a + c) + i(b + d)$$

Sottrazione

$$(a + ib) - (c + id) = (a - c) + i(b - d)$$

Moltiplicazione

$$(a + ib)\ (c + id) = (ac - bd) + i(ad + bc)i$$

Divisione

$$\frac{a + ib}{c + id} = \frac{(ac + bd)}{(c^2 + d^2)} + \frac{(bc - ad)}{(c^2 + d^2)}$$

Radice quadrata

$$x + iy = (a + ib)^{1/2}$$

donde

$$x = [a/2 + ((a/2)^2 + (b/2)^2)^{1/2}]^{1/2}$$
$$y = [-a/2 + ((a/2)^2 + (b/2)^2)^{1/2}]^{1/2}$$

Se $a > 0$ si calcola x, e poi $y = b/2x$.
Se $a < 0$ si calcola y, e poi $x = b/2y$.

Elevamento a potenza
Sappiamo che

$$(a + ib)^n = r^n e^{in\theta}$$

cioè

$$(a + ib)^n = r^n (\cos n\theta + i \operatorname{sen} n\theta)$$

con

$$r = (a^2 + b^2)^{1/2}$$
$$\theta = \operatorname{arcotg} b/a$$

2. Scrivere un programma che permetta di creare un record di persone con alcuni campi per lo stato civile (coniugato, celibe, etc.) ed altri per gli eventuali figli (senza figli, 1, 2, etc.) con l'età, il sesso ed il nome di battesimo relativi.

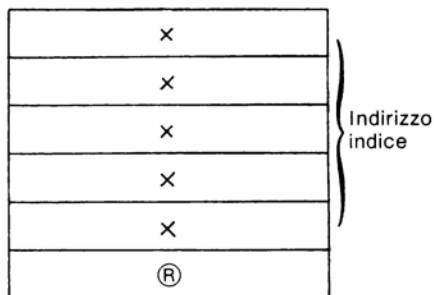
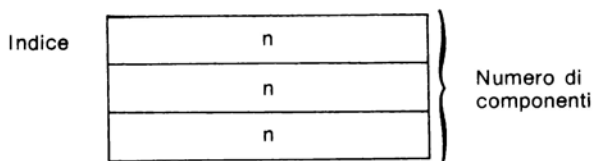
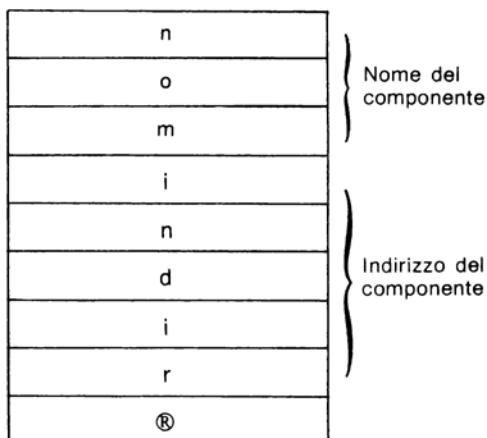
3. Scrivere un programma che permetta di distribuire delle mani in un gioco di carte. Si potrà usare la funzione *random*(*x*) di cui si parlerà nel Capitolo 9.

4. Scrivere un programma che permetta d'inserire il carattere di controllo ogni volta che un flusso viene trasmesso su una stampante.

5. Scrivere un programma che calcoli la frequenza dei caratteri in un flusso/testo, per mezzo delle procedure *get* e *write*.

6. Modificare i programmi presentati nel Capitolo in modo che si possano effettuare un'aggiunta ed una modifica.

7. Scrivere un programma che crei un indice in un flusso sequenziale di tipo testo, assumendo la prima linea del flusso-testo se-

quenziale come un numero di cinque cifre (da 0 a 99999) che e-sprime l'indirizzo dell'indice. Si abbia cioè

Anche l'indice è formato da più elementi, di cui il primo rappresenta il numero dei componenti del flusso, espresso in tre cifre (da 0 a 999);

8. Scrivere un programma che scriva un nuovo componente in un flusso indicizzato organizzato sequenzialmente. Ogni entrata nell'indice è formata dal nome del componente espresso in tre caratteri seguito dall'indirizzo iniziale del componente, espresso in cinque cifre: il nome del componente può essere un numero di tre cifre.

9. Scrivere un programma che legga un componente di un flusso indicizzato.

10. Creare un flusso di dati relativi a società commerciali e industriali (*s.c.i.*), contenente le seguenti informazioni: un numero d'identificazione, il numero dei dipendenti, il passivo o l'attivo dell'anno precedente, l'esistenza di contratti pubblici, l'esistenza di sovvenzioni statali.
Il flusso *s.c.i.* è un flusso di records: sono richieste semplicemente le dichiarazioni e le istruzioni d'ingresso/uscita necessarie per leggere/scrivere i records del flusso.

11. Il Ministero dell'Industria e Commercio s'interessa alle società commerciali ed industriali (s.c.i.) in difficoltà, ed ha stabilito una soglia (s) in base alla quale si possa definire il carattere di gravità della crisi dell'impresa, mediante il rapporto (deficit)/(numero dei dipendenti).
Se $d/n > s$, l'impresa è in *serie* difficoltà.
Bisogna ottenere le seguenti informazioni:
— Trovare le s.c.i. con meno di 500 dipendenti in serie difficoltà, e sovvenzionate dallo Stato.
— Trovare le s.c.i. con più di 500 dipendenti in serie difficoltà, e con contratti pubblici e sovvenzioni statali.
— Trovare le s.c.i. in serie difficoltà prive sia di contratti pubblici che di sovvenzioni.
— Trovare le s.c.i. in difficoltà (di qualunque portata) che abbiano ottenuto contratti pubblici, ma non sovvenzioni.

a) Esprimere i quesiti precedenti sotto forma di domande logiche, cioè di espressioni booleane.

b) Qual è la più semplice domanda che permetta di ottenere in una sola volta tutte le informazioni desiderate?

c) Scrivere un programma in Pascal che permetta di selezionare in una sola volta tutte le informazioni richieste.

CAPITOLO 7

I PUNTATORI E LE STRUTTURE DI DATI DINAMICHE

"Non cerchiamo quindi certezza e stabilità. La nostra ragione è sempre delusa dall'incostanza delle apparenze, niente può fermare il finito fra i due infiniti che lo racchiudono e lo fuggono".

PASCAL,
Pensées

Il problema della definizione e del trattamento delle strutture di dati è un problema centrale dell'informatica. Le strutture di dati sono degl'insiemi di dati messi in relazione da strutture. Queste relazioni possono essere di natura sintattica (determinati oggetti strutturati secondo regole definite *a priori*) o di natura semantica (un gruppo di oggetti con un rapporto comune, ossia un insieme).

Nel Capitolo 4 abbiamo dato degli esempi di queste strutture con vettori e matrici, nei quali i dati hanno una relazione di posizione associata alla loro funzione. Un esempio più complesso potrebbe essere l'indice di una biblioteca: ad un libro si accede in base al titolo, all'autore o all'argomento, presenti in un elenco di parole-chiave. Tutte queste strutture sono più o meno efficaci secondo il tipo di ricerca che si conduce.

In questo capitolo studieremo le strutture di dati dinamiche, gestite per mezzo di *puntatori*.

È fondamentale conoscere le strutture di dati, perchè, a seconda dell'applicazione e delle risorse di cui si dispone, saremo in grado di optare per l'una piuttosto che per l'altra. Così, ad esempio, la quantità di memoria necessaria per codificare gli stessi dati, ma diversamente strutturati, può cambiare di un fattore 10. E analogamente, per

quanto riguarda il tempo di elaborazione, con strutture diverse si possono avere tempi che vanno, per un insieme di *n* dati, da *n!* ad *n log n*.

Quindi in questo capitolo esamineremo alcune strutture di dati, e come si realizzano tramite un meccanismo di puntatori.

1 - LE STRUTTURE DI DATI

1.1 - I concetti d'indirizzo e di puntatore

Quando i dati sono memorizzati in un supporto di memoria, il solo modo per accedervi è conoscerne l'esatta posizione, oppure l'esatto contenuto. (Ad esempio, in una biblioteca si accede ad un libro conoscendone la posizione, trascritta anche sulla costola).

Una volta che si conosca la collocazione fisica del dato, questo viene associato ad un numero intero, che corrisponde alla zona di memoria fisica in cui si trova il dato. Questo numero è l'*indirizzo* del dato, che può venir associato ad una parola-memoria, ad un carattere o ad un blocco di memoria (pagina della memoria centrale, settore di un disco magnetico).

In un programma, generalmente gl'indirizzi sono indirizzi simbolici, cioè individuati da nomi, che vengono poi tradotti in indirizzi fisici all'esecuzione del programma.

1.2 - I vari tipi d'indirizzamento

— Indirizzamento *diretto*: un dato corrisponde ad un indirizzo reale, o *effettivo*.
— Indirizzamento *relativo*: in questo caso si ha a che fare con un indirizzamento relativo ad un indirizzo corrente. Dire, ad esempio, che un dato si trova ad un indirizzo *y*, relativo ad *x* (indirizzo corrente), vuol dire che l'indirizzo reale, o effettivo, si ottiene sommando *x* ad *y*.
— Indirizzamento *indicizzato*: un dato è associato ad un indirizzo *x* e ad un indice *i*. In un insieme di dati, individuato da un indirizzo d'inizio *x*, si accederà ad un dato specifico per mezzo di un indice indicante la posizione dell'elemento rispetto ad *x*. L'indirizzo *i* si trova generalmente in un registro indice dell'unità centrale.
— Indirizzamento *indiretto*: un dato è associato ad un indirizzo 1, associato a sua volta ad un indirizzo 2: all'indirizzo 1 si trova non già il dato, ma un secondo indirizzo, che è l'indirizzo effettivo, o reale. Si può avere un indirizzamento indiretto a più livelli.

1.3 - I puntatori

Possiamo definire i puntatori come dati il cui *contenuto è un indirizzo*.

Così, nel caso dell'indirizzamento indiretto, il contenuto del primo indirizzo può esser visto come un puntatore, perchè rimanda all'indirizzo effettivo di un dato.

In informatica l'impiego dei puntatori è estremamente comodo, perchè permette di definire delle strutture di dati relativamente complesse.

1.4 - Gl'indici

Un indice è un dato che permette di specificare *un indirizzo relativo* in una struttura di dati. Si tratta di un concetto diverso da quello di puntatore: la differenza è che un indice individua sempre *direttamente* un elemento di un insieme (l'ennesimo elemento), mentre un puntatore individua *indirettamente* un dato o un insieme di dati.

Presentiamo ora le principali strutture di dati ed esaminiamo più particolarmente le operazioni dinamiche eseguibili in Pascal su queste strutture, e cioè:

- accesso ad un elemento qualsiasi;
- inserzione di un nuovo elemento;
- cancellazione del k-esimo elemento;
- copia della struttura e dei dati;
- combinazione di più strutture dello stesso tipo;
- scomposizione di una struttura in più strutture dello stesso tipo

2 - LE LISTE LINEARI

Sono le strutture di dati più semplici, nelle quali i dati sono organizzati in maniera lineare.

2.1 - Lista lineare semplice

Una lista lineare è un insieme di elementi indicati con $x(1)$; $x(2)$, ... $x(n)$, la cui struttura si basa soltanto sulle posizioni relative degli elementi: infatti ciascun elemento è individuato da un indice i.

2.1.1 - Allocazione di memoria

Una lista lineare è caratterizzata dal fatto che tutti i dati sono memorizzati in sequenza sul relativo supporto di memoria (memoria centrale o ausiliaria). Così, se ciascun elemento corrisponde ad una parola-memoria, e se la lista x comincia all'indirizzo a, gli elementi successivi $x(2)$, $x(3)$, ...saranno memorizzati in $a + 1$, $a + 2$, ... $a + n$. Questa struttura è definita in Pascal con le dichiarazioni di *vettore* (v. Cap. 4).

Esistono due tipi di allocazione:

- L'allocazione *fissa*, nella quale la dimensione della lista vettore viene fissata per mezzo di una dichiarazione in fase di compilazione. Questo pone dei limiti, ma è sempre possibile scegliere per i vettori una dimensione tale da coprire tutti i casi possibili.

– L'allocazione *dinamica*, disponibile in alcuni linguaggi evoluti, come l'ALGOL. Permette di definire per i vettori delle dimensioni variabili. In Pascal non è disponibile per i vettori.

2.1.2 - Accesso ad un elemento qualsiasi

L'accesso ha luogo per mezzo di un indice *i*, che permette di ottenere l'elemento *a(i)*.

Come si è visto, in Pascal la modalità di accesso viene definita per mezzo della dichiarazione di VETTORE (ARRAY), e l'elemento *a(i)* rappresenta l'i-esimo elemento del vettore.

2.1.3 - Inserzione di un elemento

Si realizza con un meccanismo molto semplice, perchè basta operare in questo modo:

– determinare il valore dell'indice *i* nel punto in cui va fatta l'inserzione;
– far slittare di un posto, verso destra, gli elementi da *a(i)* ad *a(n)*;
– inserire il nuovo elemento *a(i)*;
– incrementare *n* di 1.

Questo può far sorgere dei problemi nei linguaggi che non prevedono l'allocazione dinamica: infatti, essendo la dimensione del vettore fissata a priori con una dichiarazione, occorre prevedere la dimensione massima che il vettore assumerà.

Inoltre l'inserzione potrà richiedere un tempo di elaborazione molto elevato, perchè, in media, bisogna spostare la metà degli elementi.

2.1.4 - Cancellazione di un elemento

La cancellazione è l'operazione inversa della precedente, e quindi si effettua in modo analogo:

– determinare il valore dell'indice *i* dell'elemento da cancellare;
– far slittare di un posto, verso sinistra, tutti gli elementi da *a(i + 1)* ad *a(n)*;
– decrementare *n* di 1.

Qui il problema dell'allocazione fissa non si pone più, perchè la dimensione del vettore diminuisce.

2.1.5 - Combinazione di due liste lineari semplici

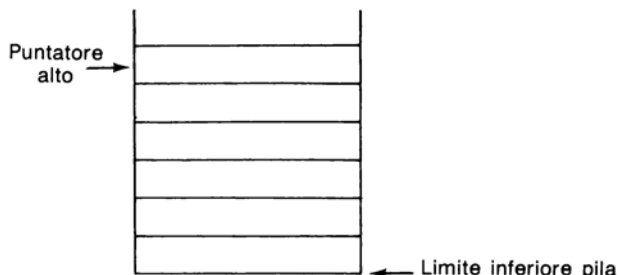
Si realizza ripetendo l'algoritmo d'inserzione di un elemento. Pertanto valgono le osservazioni svolte a questo proposito.

2.1.6 - Copia di una lista lineare semplice

L'algoritmo è immediato.

2.2 - La struttura a pila

Si tratta di una struttura a lista lineare nella quale inserzioni e cancellazioni avvengono da una sola estremità, cioè dall'alto della pila.

Quindi una pila è rappresentabile mediante lo schema che segue. Essendo variabile la dimensione, cioè l'altezza, di una pila, è necessario che un puntatore, o un indice, ne indichi la cima.

Le operazioni che permettono di aggiungere o di togliere un elemento sono spesso dette d'inserimento e di estrazione: alcuni calcolatori hanno queste istruzioni cablate.

Si parla anche di struttura LIFO (dall'inglese Last In First Out, cioé ultimo arrivato primo uscito).

Queste strutture sono utili per memorizzare dei dati in forma temporanea: ad esempio si usano molto nell'analisi sintattica o nella compilazione delle frasi di un linguaggio, durante la quale s'inseriscono l'uno nell'altro i vari elementi lessicali della frase via via che si trovano, per determinare la struttura sintattica della frase.

2.2.1 - Copia di una pila

Per copiare una pila si copiano tutti gli elementi compresi fra l'elemento indirizzato da *puntatore alto* e quello indirizzato da *limite inferiore pila*. La differenza fra i valori di questi due puntatori permette di calcolare il numero degli elementi contenuti nella pila.

2.2.2 - Allocazione di memoria

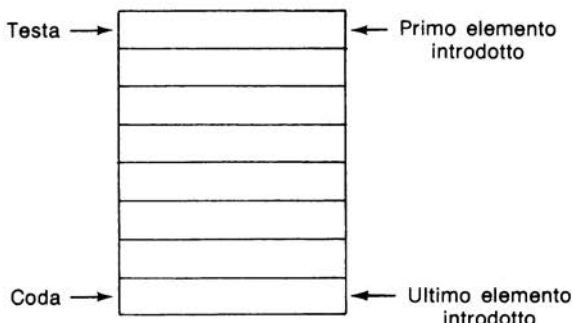
Con le strutture a pila ha un peso determinante il problema del superamento delle dimensioni. Con l'allocazione statica, il problema è risolvibile solo con uno spazio di riserva, nel quale verranno disposti gli elementi in eccedenza: il problema comunque può tornare a porsi per la riserva. Con un sistema a più pile, studi statistici permettono di determinare un "*pool*" di riserva che garantisce di non trovarsi di fronte al pro-

blema, se non in situazioni veramente eccezionali (saturazione del calcolatore, guasto, etc.).

2.2.3 - Fusione di due pile

La fusione di due pile è un problema che si presenta molto raramente, ed ha senso solo se si aggiunge a ciascun dato il tempo, o momento, nel quale è stato introdotto nella pila, in modo da rispettare la regola dell'"ultimo arrivato primo uscito".

2.3 - Le code, o file d'attesa

Le code sono liste lineari nelle quali gli elementi vengono aggiunti o tolti ad uno ad uno, ma secondo un meccanismo diverso da quello che vale per le pile. Vige infatti la regola FIFO (dall'inglese First In First Out, cioè primo arrivato primo uscito). Pertanto questa struttura costituisce una fila d'attesa le cui estremità possono essere rappresentate da due puntatori, detti *testa* e *coda*.

In una struttura di questo tipo tutte le inserzioni avvengono tramite il *puntatore coda*, e tutti i prelievi tramite il *puntatore testa*.

Si tenga presente che, quando il puntatore coda raggiunge il valore massimo, può essere azzerato di nuovo, perchè si suppone che nel frattempo il puntatore testa sia andato avanti: se i puntatori testa e coda si congiungono, vuol dire che la fila d'attesa è piena. Questo è quello che chiamiamo fila d'attesa circolare.

Analogamente, quando il puntatore testa raggiunge il valore massimo, viene azzerato di nuovo: se i puntatori testa e coda sono uguali, vuol dire che la fila d'attesa è vuota.

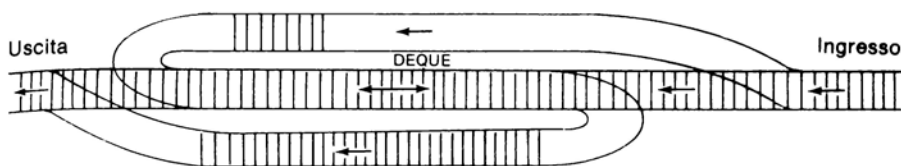
Le file d'attesa sono molto usate in quelle applicazioni software nelle quali ci sono diversi utilizzatori che fanno la coda davanti ad una stessa risorsa. Questa struttura si può usare anche per elaborazioni di dati via via che i dati arrivano.

La combinazione di due file d'attesa è molto rara; per realizzarla, bisogna associare a ciascun dato l'ora del suo arrivo.

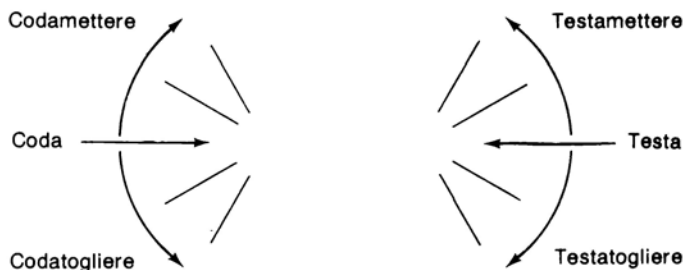
2.4 - La fila d'attesa a doppio ingresso e doppia uscita

È una struttura di dati in cui gli elementi possono venir introdotti o tolti dalle due estremità della fila. Prende anche il nome di DEQUE, dall'inglese Double Ended QUEue, cioè coda a due terminazioni.

Questa struttura è così schematizzabile:

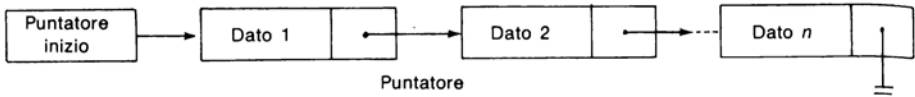
Se quest'anello manca, avremo una DEQUE limitata in ingresso

Se quest'anello manca, avremo una DEQUE limitata in uscita

Si tratta di fatto di una generalizzazione di una fila d'attesa (coda) e di una pila: infatti, se si utilizza la via diretta d'ingresso e la via di uscita dal basso, si ottiene una pila; se, per converso, si utilizza la via d'ingresso dall'alto e la via di uscita dal basso, si ha una coda.

Si può anche schematizzare una DEQUE facendo riferimento ad una lista circolare:

I puntatori testa (*testamettere* e *testatogliere*) si spostano lungo le direzioni indicate quando si mette o si toglie un elemento dalla testa della DEQUE; analogamente per quanto concerne i puntatori coda. Allora, se si usa *codamettere* (ingresso dall'alto, nello schema precedente) e *testatogliere* (uscita dal basso, nello schema precedente), si ha una fila d'attesa normale. Se invece si usano *testamettere* e *testatogliere*, si ha una pila; lo stesso accade con *codamettere* e *testatogliere*.

Attenzione: le strutture viste finora sono dinamiche soltanto in ragione del carattere dinamico del loro contenuto. Comunque la struttura di base è sempre quella di un vettore disposto in memoria in forma sequenziale, e associato ad indici che individuano ciascuno un elemento.

2.5 - Le liste concatenate

Invece che ad un'allocazione sequenziale della memoria, si può ricorrere ad un'allocazione più flessibile, che associ a ciascun dato *un puntatore* riferito all'elemento successivo: questo tipo di allocazione viene anche detto allocazione concatenata.

Questa è una lista lineare semplice, rappresentata in modo concatenato: è opportuno introdurre un *puntatore inizio*, corrispondente al primo dato.

2.5.1 - Confronto fra i due tipi di struttura

Dal punto di vista dello spazio di memoria, è evidente che la rappresentazione concatenata occupa uno spazio maggiore, perchè vengono memorizzati non solo i dati, ma anche i puntatori.

Se i dati sono poco ingombranti, ad esempio una parola-memoria, e lo spazio per l'indirizzamento è esteso (approssimativamente una parola-memoria), il coefficiente di utilizzo è del 50%. Se invece i dati sono voluminosi, questa tecnica diventa conveniente. Quindi adotteremo di preferenza l'allocazione concatenata quando il rapporto (dimensione di un dato)/(dimensione di un indirizzo) è elevato.

Questo tipo di struttura è realizzabile in Pascal grazie al concetto di puntatore.

3 - I PUNTATORI IN PASCAL

Le variabili che abbiamo esaminato finora erano puramente statiche. In particolare, si è visto che le dimensioni dei vettori devono essere fissate quando i vettori vengono dichiarati.

Il concetto di puntatore, in Pascal, permette di creare delle strutture dinamiche. Vedremo che, per mezzo di variabili puntatore, si possono creare e manipolare strutture più complesse, come ad esempio le strutture ad albero.

3.1 - Definizione del tipo puntatore

Dal punto di vista sintattico, un tipo puntatore è definito da un identificatore di tipo preceduto da una freccia.

Tipo puntatore

Così, ad esempio, si può avere

```
TYPE
  puntatore =↑dato;
  dato = ARRAY[1 ... 10] OF integer;
```

Qui il tipo puntatore fa *riferimento* ad un vettore di dati interi: il "riferimento" consiste, molto semplicemente, in un indirizzo di memoria. Infatti un puntatore è un dato che include un indirizzo! Esso permette di riferirsi a strutture di dati complesse, che vanno dal semplice tipo scalare al tipo record. Quindi si può avere altrettanto bene

```
TYPE
  puntatoreintero = ↑integer;
```

e

```
TYPE
  puntatorecomponente = ↑componente;
  componente = RECORD nomeprodotto: string;
                      quantita: integer;
                      prezzo: real
               END;
```

È anche possibile che fra gli elementi che compongono un record ci siano degli elementi di tipo puntatore. Così, nell'esempio precedente, potremmo avere

```
componente = RECORD nome: string;
                    quta: integer;
                    prezzo: real;
                    equivalente: puntatorecomponente
             END;
```

3.2 - Le variabili puntatore

Il tipo puntatore che si è appena visto permette di definire a sua volta delle *variabili puntatore*, che sono, per l'esattezza, dei puntatori.

Esempi

```
TYPE
  rif = ↑num;
  num = RECORD numero: integer;
               legame: rif
        END;
VAR
  puntatore: rif;
```

Avremmo anche potuto scrivere direttamente

```
VAR
  puntatore: ↑num;
```

Qui *rif* è un tipo puntatore, *puntatore* è una variabile puntatore.

Anche qui il tipo puntatore rappresenta un insieme di valori che puntano ad elementi di un certo tipo, mentre la variabile puntatore contiene, ad un istante dato, un valore che fa riferimento ad una variabile del tipo associato: questa variabile associata è espressa da *puntatore↑*, cioè, generalmente, dall'identificatore della variabile puntatore seguito da una freccia.

L'identificatore *puntatore↑* è del tipo associato *num* che, nell'esempio precedente, è un record. Quindi possiamo scrivere

```
puntatore↑.numero: = 10
```

che vuol dire che il campo *numero* del record *num* puntato dalla variabile *puntatore* ha il valore 10.

Consideriamo, come altro esempio, la dichiarazione

```
var rif: ↑integer;
```

rif è una variabile puntatore, *rif↑* è una variabile intera alla quale si potranno assegnare valori di espressioni intere.

Invece, in

```
TYPE
  avo: ↑genitore
  genitore = RECORD generazione: integer;
                    padre: avo;
                    nomepadre: string;
                    madre: ava;
                    nomemadre: string
                END;
VAR
  ascendente: avo
```

ascendente è una variabile puntatore, *ascendente↑* una variabile di tipo *genitore* (che è il record definito nell'esempio).

Ma qui *ascendente↑.padre* è un puntatore che si riferisce ai nonni paterni, mentre *ascendente↑.padre↑* è una variabile che contiene le informazioni relative ai nonni.

3.3 - Il valore NIL

Quando una variabile puntatore non si riferisce a nessun elemento del tipo associato, il suo valore è definito dalla parola NIL, che è una costante simbolica specifica per questo caso particolare. Il valore NIL può essere assegnato ad una qualunque variabile di tipo puntatore, e può comparire anche in espressioni condizionali relative a variabili puntatore.

In particolare, questo valore permette di verificare i limiti di una struttura di dati (fine di una lista concatenata, fine di un albero, etc.).

Avremo quindi programmi così strutturati:

```
WHILE puntatore < > NIL ...
```

oppure:

```
REPEAT
........
UNTIL puntatore = NIL
```

Analogamente, per avviare la creazione di una struttura dinamica, avremo:

```
puntatore: = NIL
```

3.4 - Le strutture dinamiche

Abbiamo visto che, per definire un puntatore, è sufficiente definire una sola variabile: in fase di compilazione viene riservato solo lo spazio necessario al puntatore definito. In fase di esecuzione, invece, è possibile richiedere l'allocazione di nuovi elementi in modo dinamico.

3.5 - La procedura new

L'operazione di allocazione di un nuovo elemento si effettua per mezzo di una procedura standard. Questa procedura, che è *new* (cioè *nuovo*), ha il seguente formato di chiamata:

```
new (puntatore);
```

Il parametro è una variabile puntatore. La procedura permette di assegnare lo spazio necessario ad una nuova variabile, del tipo associato al puntatore (*puntatore↑*), e di assegnare alla variabile puntatore l'indirizzo di memoria dello spazio assegnato.

È possibile costruire strutture concatenate più complesse, se la variabile puntatore fa riferimento ad un record che contenga a sua volta un elemento di tipo puntatore.

Possiamo usare la procedura *new* anche per creare una nuova variabile tale per cui il tipo ad essa associato sia un record con campi variabili: in questo caso lo spazio è sufficiente per tutti i campi.

La procedura *new* ha anche un'altra forma, che è

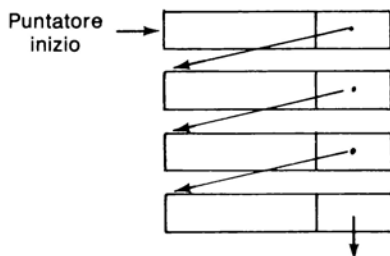
new (puntatore, c1, c2, ... cn)

ove i parametri *ci* sono dei valori che corrispondono alle alternative di una struttura CASE, dichiarata in un record a campi variabili. In tal modo si possono riservare degli spazi per tutti i campi variabili contenuti nell'elenco dei parametri.

I parametri *ci* devono essere elencati nell'ordine definito dalla dichiarazione; i campi che variano oltre l'ultimo parametro *ci* saranno omessi.

La variabile associata non deve cambiare di campo durante l'esecuzione del programma; inoltre non è possibile assegnare globalmente dei valori alla variabile associata.

3.5.1 - Applicazione della lista concatenata alle strutture a pila

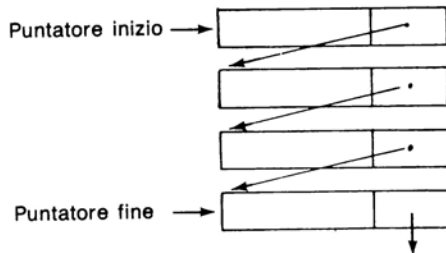
Una pila può essere rappresentata con una lista concatenata nel modo seguente:

a) *Inserzione di un elemento sulla sommità della pila*
   Viene creato un nuovo *puntatore alto* per il nuovo dato; il vecchio *puntatore inizio* diventa il puntatore di questo dato al precedente elemento della sommità della pila.

b) *Estrazione di un elemento dalla sommità della pila (lettura)*
   Il puntatore di collegamento del dato sulla sommità della pila diventa il puntatore inizio.

Avremo il programma seguente:

```
PROGRAM pilaconcatenata;
TYPE
  ptr = ↑dato;
  dato = RECORD numero: integer;
                ptrnum: ptr
         END;
```

```
VAR
  pt, ptrinizio, puntatore: ptr;
  n: integer;
BEGIN
  (*creazione della struttura concatenata*)
  ptrinizio: = NIL;
  WHILE n > 0 DO
    BEGIN
      read (n);
      new (puntatore);
      puntatore↑.numero: = n;
      puntatore↑.ptrnum: = ptrinizio;
      ptrinizio: = puntatore
    END;
  (*lettura della struttura concatenata*)
  pt: = ptrinizio;
  WHILE pt↑.ptrnum <> NIL DO
    BEGIN
      write (pt↑.numero, ' ');
      pt: = pt↑.ptrnum
    END;
  write (pt↑.numero)
END.
```

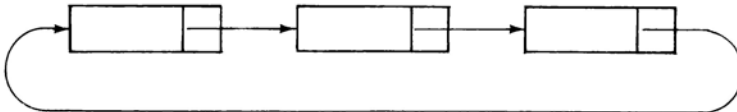
3.5.2 - Rappresentazione di una fila d'attesa con una lista concatenata

a) *Aggiunta di un elemento nella fila*
Il puntatore fine diventa il puntatore al nuovo dato, e il puntatore dell'ultimo dato diventa il puntatore di collegamento al nuovo dato.

b) *Estrazione di un elemento della fila*
Il puntatore inizio diventa il puntatore al secondo elemento, e così via.

Una struttura di questo tipo è realizzata nel programma che segue:

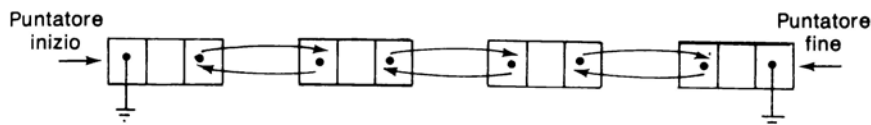
```
PROGRAM codaconcatenata;
TYPE
  ptr = ↑dato;
  dato = RECORD numero: integer;
                ptrnum: ptr
         END;
VAR
  pt, ptrinizio, puntatore: ptr;
  n: integer;
BEGIN
  (*creazione della struttura concatenata*)
  new (puntatore);
  n: = 1;
  ptrinizio: = puntatore;
  WHILE n > 0 DO
    BEGIN
      read (n);
      puntatore↑.numero: = n;
      pt: = puntatore;
      new (puntatore);
      pt↑.ptrnum: = puntatore
    END;
  pt↑.ptrnum: = NIL;
  (*lettura della struttura concatenata*)
  pt: = ptrinizio;
  WHILE pt↑.ptrnum <> NIL DO
    BEGIN
      write (pt↑.numero, ' ');
      pt: = pt↑.ptrnum
    END;
  write (pt↑.numero)
END.
```

3.5.3 - Le liste circolari

Una lista circolare è una lista concatenata nella quale il puntatore dell'ultimo elemento punta al primo elemento:

Qui il concetto di puntatore d'inizio non ha più alcun senso, perché la lista è percorribile partendo da un punto qualunque. Questo metodo è utile per rappresentare le strutture d'insiemi.

3.5.4 - Le liste a doppia catena

Volendo avere una flessibilità ancora più elevata, definiremo delle liste che prevedono due tipi di puntatori:
- puntatori in avanti;
- puntatori all'indietro.

Questa struttura è rappresentabile come segue:

In tal modo si hanno nello stesso tempo una struttura a pila ed una a fila d'attesa.

Quindi dovremo fondere insieme i due programmi precedenti. A seconda che la lettura della struttura avvenga nell'uno o nell'altro senso, avremo una pila o una fila d'attesa. Il programma così ottenuto sarà il seguente:

```
PROGRAM codapila;
TYPE
   ptr = ↑dato;
   dato = RECORD numero: integer;
                 avanti: ptr;
                 indietro: ptr
          END;
VAR
   pt, ptrinizio, ptrfine, puntatore: ptr;
   n: integer;
BEGIN
   (*creazione della struttura concatenata*)
   new (puntatore);
   n: = 1;
   ptrfine: = nil;
   ptrinizio: = puntatore;
   WHILE n > 0 DO
      BEGIN
         read (n);
         puntatore↑.numero: = n;
         puntatore↑.avanti: = ptrfine;
```

```
      ptrfine: = puntatore;
      pt: = puntatore;
      new (puntatore);
      pt↑.indietro: = puntatore;
    END;
  pt↑.indietro: = NIL;
  (*lettura della struttura concatenata*)
  (*in avanti = pila*)
  pt: = ptrfine;
  WHILE pt↑.avanti <> NIL DO
    BEGIN
      write (pt↑.numero, ' ');
      pt: = pt↑.avanti
    END;
  write (pt↑.numero);
  (*all'indietro = coda*)
  pt: = ptrinizio;
  WHILE pt↑.indietro <> NIL DO
    BEGIN
      write (pt↑.numero, ' ');
      pt: = pt↑.indietro
    END;
  write (pt↑.numero);
END.
```

3.6 - Accesso ad un elemento qualsiasi

Per accedere ad un elemento d_k di una lista concatenata, ci serviremo dei puntatori p_0, p_1, ... p_{k-1}.

L'accesso casuale ad un elemento della lista richiede, in media, che venga letta la metà dei puntatori: *quindi la struttura mal si adatta all'accesso casuale*. Una lista semplice con indice è molto più appropriata! Una lista concatenata si adatta invece perfettamente alle elaborazioni sequenziali.

3.6.1 - Fusione o separazione di liste concatenate

Quando non si debba eseguire un ordinamento, due liste concatenate possono essere fuse insieme molto facilmente sostituendo l'ultimo puntatore della prima lista con il puntatore inizio della seconda.

Analogamente, la divisione di una lista in due liste separate si ottiene "rompendo" la catena e sostituendo l'ultimo puntatore della prima lista con il valore NIL.

Il puntatore corrispondente viene recuperato come puntatore inizio della seconda lista.

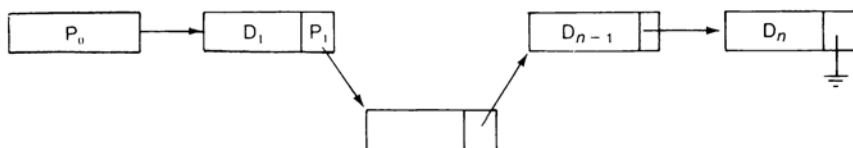
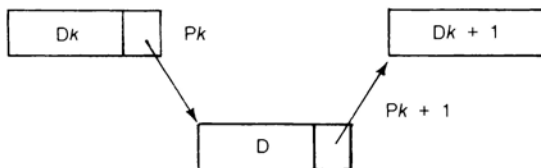
3.6.2 - Osservazioni sulle strutture concatenate e sul problema del superamento delle dimensioni

Le strutture concatenate offrono il vantaggio di utilizzare il massimo della memoria disponibile per realizzare pile o liste. Infatti, all'inizio tutta la memoria disponibile può essere considerata come una fila costituente un pool di riserva: ogni volta che si ha bisogno di un nuovo elemento, si riduce il pool di un'unità, e si unisce l'elemento in questione alla pila o alla lista, che in tal modo diventa più lunga.

Analogamente, quando si toglie un elemento dalla struttura concatenata, quest'elemento è ritornato nella lista della memoria disponibile.

Il problema del superamento delle dimensioni si pone soltanto quando la memoria utilizzabile è completamente occupata dal punto di vista fisico.

3.6.3 - Inserzione e cancellazione di un elemento

Uno dei principali vantaggi di una struttura concatenata è la facilità con la quale avvengono l'inserzione e la cancellazione di un elemento della lista.

Una volta determinata la collocazione del dato, per inserire il nuovo dato non è necessario far slittare i dati di un posto. Ad esempio, se p_0, p_1, ... p_{n-1} sono i puntatori relativi ai dati d_1 ... d_n, per inserire il dato d fra i dati d_k e d_{k+1}, bisogna richiedere l'allocazione di un nuovo elemento con un puntatore $p(d)$ tale per cui $p(d) = p_{k+1}$, cioè che punta a d_{k+1}, essendo inoltre $p(d_k) = p_k$, con p_k che punta a d.

Quest'operazione, schematizzata nella figura precedente, consiste nel cercare prima di tutto l'elemento da inserire, e poi unire il nuovo elemento per mezzo di una manipolazione dei puntatori.

Tutto il complesso di queste operazioni è eseguito dalla procedura *inserire* nel pro-

gramma che segue: il parametro *dopo* indica l'elemento della struttura dopo il quale avverrà l'inserzione. La procedura di ricerca deve tener conto del fatto che il parametro *dopo* può non comparire nella catena. Quindi è necessario sottoporre a verifica la fine della catena.

Si tenga presente che la struttura WHILE utilizzata non può avere come condizione l'espressione logica

```
(pntr↑.numero <> dopo) AND (pntr <> NIL)
```

perché, quando *pntr* è effettivamente uguale a NIL, l'elemento *pntr↑.numero* non esiste, per cui in fase di esecuzione ci sarebbe inevitabilmente un errore.

Quindi conviene usare il booleano *trovato*, che permette di uscire dal ciclo WHILE. Se non si trova l'elemento cercato, l'uscita dal ciclo sarà realizzata in virtù della condizione *pntr* = NIL.

Il programma completo è allora il seguente:

```
PROGRAM strutturaconcatenata;
TYPE
  ptr = ↑dato;
  dato = RECORD numero: integer;
                avanti: ptr
         END;
VAR
  pntr, alto, puntatore: ptr;
  a, n: integer;
(*lettura della struttura concatenata*)
(*in avanti = pila*)
PROCEDURE leggere;
BEGIN
  pntr: = alto;
  WHILE pntr↑.avanti <> NIL DO
    BEGIN
      write (pntr↑.numero, ' ');
      pntr: = pntr↑.avanti
    END;
  write (pntr↑.numero);
END;
PROCEDURE inserire (nr, dopo: integer);
VAR
  trovato: boolean;
BEGIN
  pntr: = alto;
  trovato: = falso;
  WHILE (NOT trovato) AND (pntr <> NIL) DO
```

```
    BEGIN
      IF pntr↑.numero = dopo THEN
        trovato: = true
        ELSE pntr: = pntr↑.avanti;
    END;
  new (puntatore);
  puntatore↑.avanti: = pntr↑.avanti;
  puntatore↑.numero: = nr;
  pntr↑.avanti: = puntatore;
END;
BEGIN
  (*creazione della struttura concatenata*)
  new (puntatore);
  read (n);
  alto: = NIL;
  WHILE n > 0 DO
    BEGIN
      puntatore↑.numero: = n;
      puntatore↑.avanti: = alto;
      alto: = puntatore;
      new (puntatore);
      read (n);
    END;
  leggere;
  write ('inserire nr =');
  readln (n);
  write ('dopo?');
  readln (a);
  inserire (n, a);
  leggere;
END.
```

Esecuzione

```
1  2  4  7  8  9  Ⓡ
9  8  7  4  2  1
inserire nr = 5  Ⓡ
dopo? 7 Ⓡ
9  8  7  5  4  2  1
inserire nr = 3  Ⓡ
dopo? 4  Ⓡ
9  8  7  5  4  3  2  1
inserire nr = 0  Ⓡ
dopo? 1  Ⓡ
9  8  7  5  4  3  2  1  0
```

3.6.3.1 - Cancellazione di un elemento in una struttura concatenata

Il programma che segue realizza la cancellazione di un elemento da una struttura concatenata. Questo viene effettuato dalla procedura *togliere*.

Sottolineiamo che è indispensabile memorizzare il puntatore precedente, che punta all'elemento da cancellare.

Bisogna poi tener conto del caso particolare nel quale l'elemento da togliere occupa il primo posto nella catena: in questo caso il puntatore alto è sostituito dal puntatore associato all'elemento da togliere.

```
PROGRAM strutturaconcatenata;
TYPE
  ptr = ↑dato;
  dato = RECORD numero: integer;
                avanti: ptr
         END;
VAR
  pntr, alto, puntatore: ptr;
  a, n: integer;
(*lettura della struttura concatenata*)
(*in avanti = pila*)
PROCEDURE leggere;
BEGIN
  pntr: = alto;
  WHILE pntr↑.avanti <> NIL DO
    BEGIN
      write (pntr↑.numero, ' ');
      pntr: = pntr↑.avanti
    END;
  write (pntr↑.numero);
END;
PROCEDURE togliere (nr: integer);
VAR
  trovato: = boolean;
BEGIN
  pntr: = alto;
  trovato: = false;
  WHILE (NOT trovato) AND (pntr <> NIL) DO
    BEGIN
      IF pntr↑.numero = nr THEN
        trovato: = true
        ELSE
          BEGIN
            puntatore: = pntr;
```

```
                pntr: = pntr↑.avanti;
              END;
        END;
      IF pntr = alto THEN alto: = pntr↑.avanti
        ELSE
        puntatore↑.avanti: = pntr↑.avanti;
    END;
    BEGIN
      (*creazione della struttura concatenata*)
      new (puntatore);
      read (n);
      alto: = NIL;
      WHILE n > 0 DO
        BEGIN
          puntatore↑.numero: = n;
          puntatore↑.avanti: = alto;
          alto: = puntatore;
          new (puntatore);
          read (n);
        END;
      leggere;
      write ('togliere nr =');
      readln (n);
      togliere (n);
      leggere;
    END.
```

Esecuzione

```
    1  3  5  6  7  8  9  0 (R)
    9  8  7  6  5  3  1
    togliere nr = 6  (R)
    9  8  7  5  3  1
    togliere nr = 9  (R)
    8  7  5  3  1
    togliere nr = 1  (R)
    8  7  5  3
```

OSSERVAZIONI

Togliendo un elemento non si libera automaticamente il relativo spazio di memoria. Nella maggior parte dei sistemi Pascal standard non c'è una gestione dinamica dello spazio liberato da un elemento.

Il rapporto sul Pascal di Wirth menziona la procedura *dispose*, il cui parametro è un puntatore; il sistema Pascal U.C.S.D. impiega delle procedure analoghe, *mark* e

release, che utilizzano uno spazio strutturato a pila, per cui è possibile liberare solo l'insieme dello spazio allocato dalla precedente operazione di *mark*. Non è quindi possibile liberare lo spazio occupato da un unico elemento posto, ad esempio, a metà pila.

4 - LE STRUTTURE NON LINEARI

Le strutture non lineari più semplici sono le strutture ad albero.

Una struttura ad albero permette d'introdurre dei rapporti gerarchici fra i dati.

Sono necessarie alcune definizioni preliminari.

4.1 - Definizione di albero

Un albero A è un insieme finito di elementi, detti *nodi*, tali per cui

a) esiste un nodo particolare, detto radice R;

b) l'insieme dei nodi, esclusa la radice, è suddiviso in $m \geqslant 0$ insiemi distinti $A_1 \dots A_m$, che sono a loro volta degli alberi.

Gli alberi $A_1 \dots A_m$ sono detti sottoalberi della radice R.

Questa è una definizione *ricorsiva*, perché un albero viene definito partendo da altri alberi. In tal modo la scomposizione di un albero porta come risultato finale ad un insieme di radici (un solo elemento). I nodi terminali sono detti *foglie*.

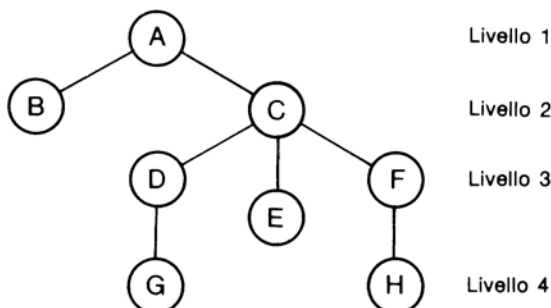
4.1.1 - Grado di un nodo

In base alla definizione data, ciascun nodo è radice di un sottoalbero. Il numero dei sottoalberi di un nodo è detto *grado* del nodo.

Un nodo terminale, cioè una foglia, ha grado zero; quindi un nodo di grado $\neq 0$ non può essere una foglia: questo viene detto talvolta nodo-ramo.

4.1.2 - Livello di un nodo

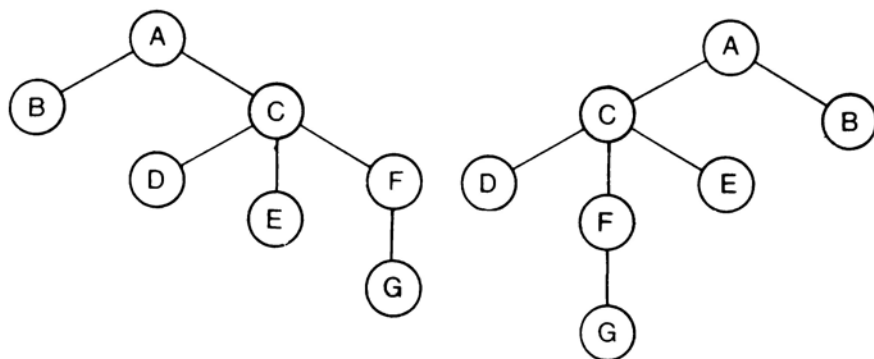
Il livello di un nodo si definisce in rapporto al livello della radice (che è 1), nel senso che è determinato in base al numero di nodi da percorrere dalla radice in poi.

Esempio

A è collegato ai due sottoalberi *B* e (*C, D, E, F, G, H*). Quindi ha grado 2.
C ha grado 3, *D* ed *F* hanno grado 1, mentre *B, G, E, H* hanno grado 0, e quindi sono foglie.

4.1.3 - Gli alberi ordinati

Se l'ordine relativo ai sottoalberi $A_1 \dots A_m$ è significativo, si dice che l'albero è *ordinato*.

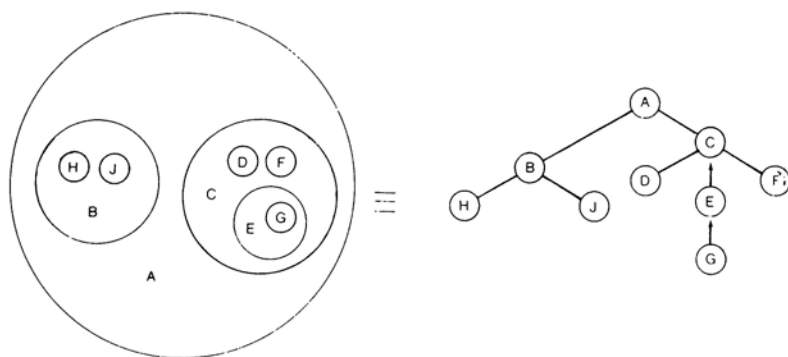
4.1.4 - Gli alberi orientati

Si dicono orientati alberi che differiscano soltanto nell'ordine dei rispettivi sottoalberi.

I due alberi che compaiono in figura sono diversi se vengono considerati come alberi ordinati; se invece sono visti come alberi orientati, sono un unico, identico albero.

Ad esempio, gli alberi genealogici sono alberi orientati, nell'ipotesi che si parta dagli antenati per arrivare ai discendenti: infatti, in essi, quel che conta è il rapporto fra genitori e figli.

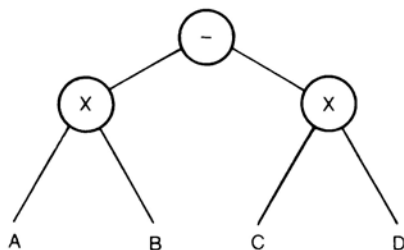
Sulla scorta di quest'esempio possiamo definire i concetti di *nodo padre*, *nodo figlio* e *nodi fratelli*. Chiameremo nodo padre una radice posta di fronte agli altri nodi dell'albero; inversamente, i nodi figli di una radice sono i nodi che si trovano immediatamente al di sotto della radice. I nodi fratelli sono tutti i nodi di uno stesso sottoalbero che si trovano al medesimo livello.

4.1.5 - Un'altra rappresentazione degli alberi orientati

Una struttura ad alberi orientati è rappresentabile anche in questo modo:

Le strutture ad alberi orientati sono utilissime in tutti i problemi di *classificazione*: si parte da un insieme, e lo si scompone in sottoinsiemi a loro volta scomponibili in sottoinsiemi più piccoli, e così via.

Ma si usano anche per rappresentare espressioni algebriche: ad esempio, l'espressione

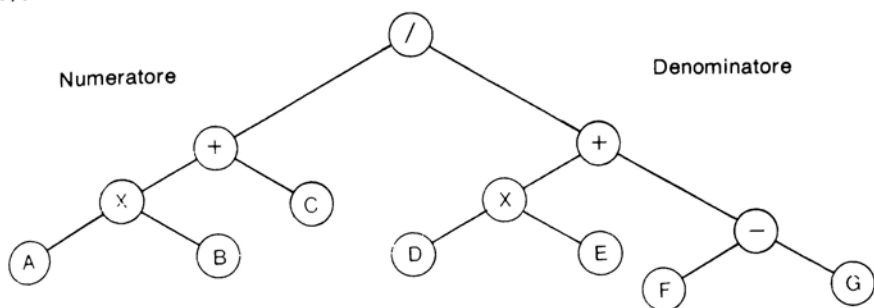
(a · b) — (c · d)

può essere così rappresentata:

Le foglie dell'albero sono le variabili, o dati, della formula, mentre i nodi di grado superiore sono gli operatori.

Un'espressione aritmetica, algebrica o logica può esser vista come una struttura ad albero nella quale i dati (o variabili) sono i nodi terminali, messi in relazione da una struttura di operatori.

Un sottoalbero dell'espressione può essere considerato come un dato, o una variabile, corrispondente ad un risultato intermedio. Ad esempio, nell'espressione

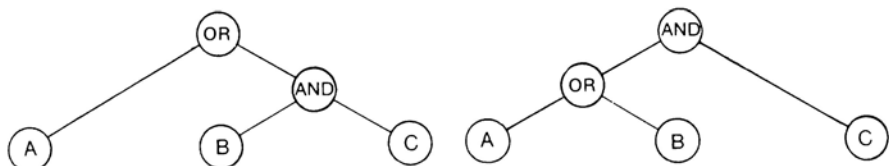
((a · b) + c)/((d · e) + (f — g))

rappresentata dall'albero

i due sottoalberi sinistro e destro rappresentano delle espressioni intermedie, che chiameremo *numeratore* e *denominatore* dell'espressione iniziale.

L'efficacia delle strutture ad albero viene dal fatto che esse permettono di istituire relazioni complesse fra dati.

In particolare, possiamo rappresentare con alberi la sintassi dei linguaggi di programmazione: il problema della traduzione (compilazione delle espressioni) è equivalente al problema della ricostruzione dell'albero che rappresenta l'espressione.

4.1.6 - Il problema dell'ambiguità

Consideriamo un'espressione logica molto semplice, come *a or b and c*. Possiamo rappresentarla in due modi:

Diciamo quindi che l'espressione, e di conseguenza la relativa norma grammaticale, è ambigua: in altre parole, un'espressione, e la norma grammaticale corrispondente, è detta *ambigua* quando è rappresentabile con più di un albero.

Questo riveste un'importanza fondamentale nel trattamento automatico delle espressioni, perchè si è di fronte ad un problema *irrisolvibile*: in altre parole, a meno di una norma ulteriore, il calcolatore non avrà gli estremi per scegliere una delle due forme ambigue.

Per questo motivo in un linguaggio di programmazione bisogna introdurre il concetto di *gerarchia*, o regola di precedenza, degli operatori.

Se questa nuova norma stabilisce che l'operazione *and* è prioritaria rispetto all'operazione *or*, allora, con riferimento all'esempio precedente, è valida solo la rappresentazione a sinistra, e l'espressione viene interpretata come

a or (b and c)

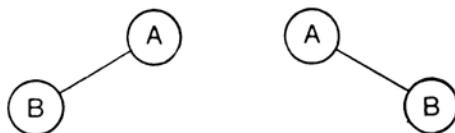
Come si vede qui, l'ambiguità può essere eliminata anche introducendo delle parentesi.

Tutti i linguaggi naturali hanno norme grammaticali ed espressioni ambigue, donde la difficoltà di trattare questi linguaggi in modo automatico. In realtà, spesso l'ambiguità è eliminata tenendo conto del contesto, ma questo richiede norme grammaticali molto più complesse ed elaborazioni più gravose; c'è però la contropartita che il calcolatore viene messo in grado di riconoscere, e segnalare, i casi di ambiguità.

4.2 - Gli alberi binari: definizione

Un albero binario è un insieme di nodi che può essere sia vuoto, sia formato da una radice e due distinti alberi binari, detti rispettivamente sottoalbero destro e sottoalbero sinistro.

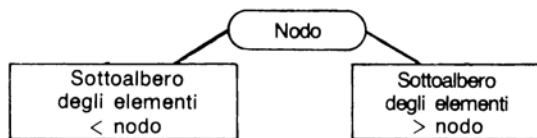
Si tratta quindi di un concetto diverso da quello di albero ordinario, non semplicemente di un caso particolare.

Ad esempio, i due alberi binari

sono diversi, perchè il sottoalbero destro del primo è vuoto, mentre, nel secondo, è vuoto il sottoalbero sinistro.

4.2.1 - Creazione di un albero binario

Possiamo creare un albero binario definendo un tipo che corrisponda ad un nodo dell'albero. Ad esempio,

```
TYPE
   puntatore = ↑nodo;
   nodo = RECORD nome: string;
                 sinistra: puntatore;
                 destra: puntatore
          END;
```

Questa struttura è utilizzabile per ordinare degli elementi, assumendo che il sottoalbero sinistro di un nodo contenga gli elementi posti *prima* dell'elemento corrispondente al nodo, e che il sottoalbero destro contenga gli elementi posti *dopo* l'elemento associato al nodo in questione:

La creazione di un albero binario è realizzata dal programma seguente, per mezzo della procedura albero.

```
PROGRAM alberobinario;
TYPE
  puntatore = ↑nodo;
  nodo = RECORD nome: string;
                sinistra: puntatore;
                destra: puntatore
         END;
VAR
  radice: puntatore;
  parola: string;
PROCEDURE albero (var rad: puntatore; elt: string);
BEGIN
  IF rad = NIL THEN
    BEGIN
      new (rad);
      rad↑.nome: = elt;
      rad↑.sinistra: = NIL;
      rad↑.destra: = NIL;
    END
    ELSE
    IF elt < rad↑.nome THEN
      albero (rad↑.sinistra, elt);
      ELSE
      albero (rad ↑.destra, elt);
END;
BEGIN
  radice: = NIL;
  write (':');
  readln (parola);
  WHILE parola < > 'fine' DO
    BEGIN
      albero (radice, parola);
      write (':');
      readln (parola);
    END;
END.
```

4.2.2 - Le procedure di "attraversamento" degli alberi

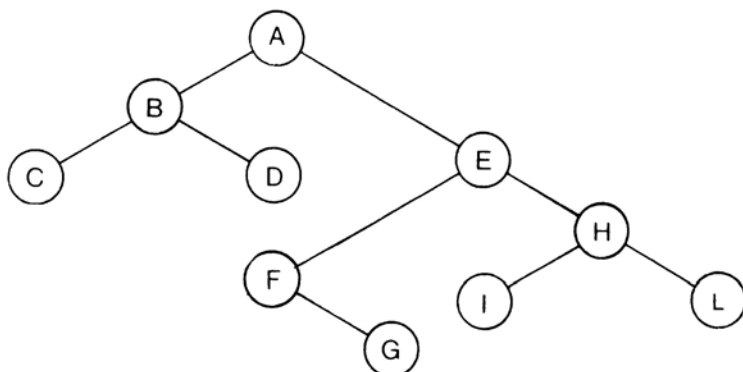
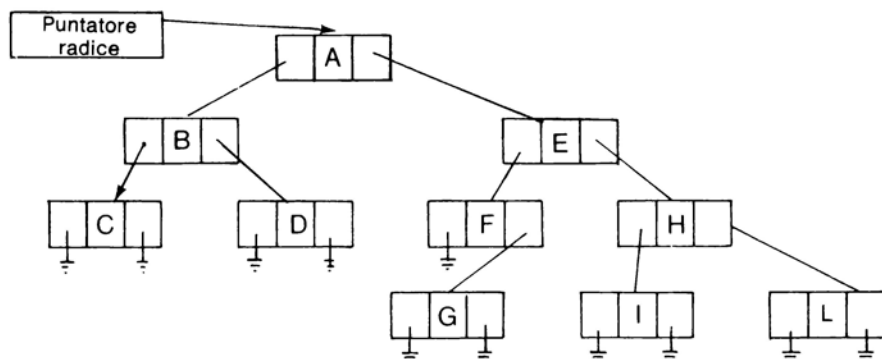
I dati possono esser contenuti nei nodi della struttura ad albero; pertanto è necessario disporre di algoritmi che permettano di "attraversare" l'insieme dei nodi quando bisogna effettuare un'elaborazione.

Attraversare una struttura di dati vuol dire percorrere questa struttura in modo tale che ciascun elemento venga esaminato una sola volta.

Esaminiamo ora l'attraversamento degli alberi binari.

Come si è detto, un albero binario è un insieme di nodi che può essere sia vuoto, sia formato da una radice e due sottoalberi binari.

Ora, ciascun nodo di un albero binario può essere rappresentato da un dato associato a due puntatori: l'uno rivolto al sottoalbero di sinistra, l'altro al sottoalbero di destra.

Così, ad esempio, l'albero binario

è rappresentabile in questo modo:

Un albero binario può essere attraversato in tre modi:

a) *Attraversamento RSD (radice - sinistra - destra)*
L'algoritmo di attraversamento è ricorsivo, ed è costituito dalle seguenti fasi:

– esame della *radice*;
– attraversamento del *sottoalbero sinistro*;
– attraversamento del *sottoalbero destro*.

Nell'esempio precedente, avremo:

A B C D E F G H I L

L'espressione "esame della radice" vuol dire passare per la radice prima di attraversare i sottoalberi che si trovano al di sotto di essa.

b) *Attraversamento SRD (sinistra - radice - destra)*
I tre passaggi avvengono in un ordine diverso:

– attraversamento del *sottoalbero sinistro*;
– esame della *radice*;
– attraversamento del *sottoalbero destro*.

Nell'esempio precedente, avremo:

C B D A F G E I H L

L'attraversamento SRD è senza dubbio il più facile da ricordare, perchè corrisponde ad una proiezione in orizzontale dei nodi, da sinistra verso destra.

c) *Attraversamento SDR (sinistra - destra - radice)*
I passaggi dell'algoritmo si susseguono in quest'ordine:

– attraversamento del *sottoalbero sinistro*;
– attraversamento del *sottoalbero destro*;
– esame della *radice*.

Sempre nel solito esempio, abbiamo:

C D B G F I L H E A

che è probabilmente l'ordine meno "naturale".

Nel programma che segue si trovano gli algoritmi relativi ai tre metodi, espressi rispettivamente dalle procedure *preordine*, *centro* e *postordine*.

```
PROGRAM alberobinario;
TYPE
  puntatore = ↑nodo;
  nodo = RECORD nome: string;
                sinistra: puntatore;
                destra: puntatore
         END;
VAR
  radice: puntatore;
PROCEDURE preordine (sottoalb: puntatore);
BEGIN
  IF sottoalb < > NIL THEN
    BEGIN
      write (sottoalb↑.nome,' ');
      preordine (sottoalb↑.sinistra);
      preordine (sottoalb↑.destra);
    END;
END;
PROCEDURE centro (sottoalb: puntatore);
BEGIN
  IF sottoalb < > NIL THEN
    BEGIN
      centro (sottoalb↑.sinistra);
      write (sottoalb↑.nome, ' ');
      centro (sottoalb↑.destra);
    END;
END;
PROCEDURE albero (var rad: puntatore);
VAR
  parola: string;
BEGIN
  readln (parola);
  IF parola = 'vuota' THEN rad: = NIL
    ELSE
      BEGIN
        new (rad);
        rad↑.nome: = parola;
        write (rad↑.nome,'sinistra:');
        albero (rad↑.sinistra);
        write (rad↑.nome,'destra:');
```

```
            albero (rad↑. destra)
          END;
END;
PROCEDURE postordine (sottoalb: puntatore);
BEGIN
  IF sottoalb < > NIL THEN
    BEGIN
      postordine (sottoalb↑. sinistra);
      postordine (sottoalb↑. destra);
      write (sottoalb↑. nome,' ');
    END;
END;
BEGIN
  albero (radice);
  preordine (radice);
  writeln; postordine (radice);
  writeln; centro (radice);
END.
```

4.2.3 - Ordinamento mediante albero binario

Se nel programma che crea un albero binario inseriamo la procedura di attraversamento centro, otteniamo un programma di ordinamento alfabetico delle parole, cioè:

```
PROGRAM alberobinario;
TYPE
  puntatore = ↑nodo;
  nodo = RECORD nome: string;
                sinistra: puntatore;
                destra: puntatore
         END;
VAR
  radice: puntatore;
  parola: string;
PROCEDURE centro (sottoalb: puntatore);
BEGIN
  IF sottoalb < > NIL THEN
    BEGIN
      centro (sottoalb↑. sinistra);
      write (sottoalb↑. nome,' ');
      centro (sottoalb↑. destra);
    END;
END;
```

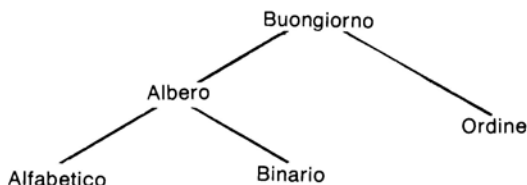
```
PROCEDURE albero (var rad: puntatore; elt: string);
BEGIN
  IF rad = NIL THEN
    BEGIN
      new (rad);
      rad↑.nome: = elt;
      rad↑.sinistra: = NIL;
      rad↑.destra: = NIL;
    END
    ELSE
    IF elt < rad↑.nome THEN
      albero (rad↑. sinistra, elt)
      ELSE
      albero (rad↑.destra, elt);
END;
BEGIN
  radice: = NIL;
  write (':');
  readln (parola);
  WHILE parola < > 'fine' DO
    BEGIN
      albero (radice, parola);
      writeln;
      centro (radice);
      write (':');
      readln (parola);
    END;
END.
```

Esecuzione

```
... Carla Ⓡ
    Carla: Marcello Ⓡ
    Carla Marcello: Dante Ⓡ
    Carla Dante Marcello: Martino Ⓡ
    Carla Dante Marcello Martino: Annamaria Ⓡ
    Annamaria Carla Dante Marcello Martino: fine Ⓡ
```

AVVERTENZA

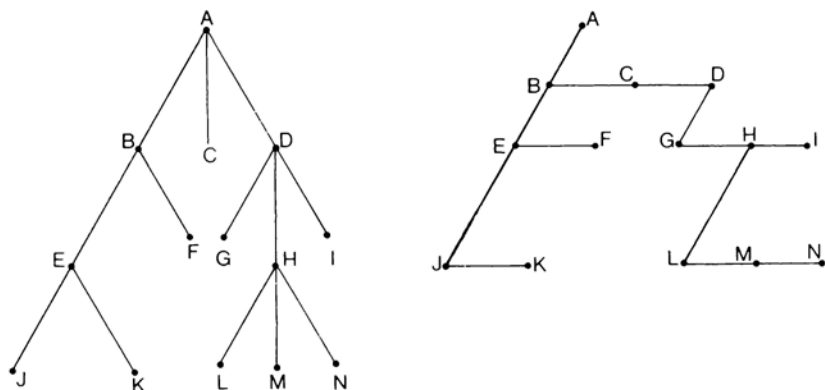
Questo è un programma di ordinamento mediante albero binario: è applicabile anche a valori numerici, cambiando il tipo del primo elemento del nodo.

Introducendo le parole *buongiorno*, *albero*, *binario*, *ordine*, *alfabetico*, otteniamo quest'albero:

4.2.4 - Rappresentazione di un albero qualunque sotto forma di albero binario

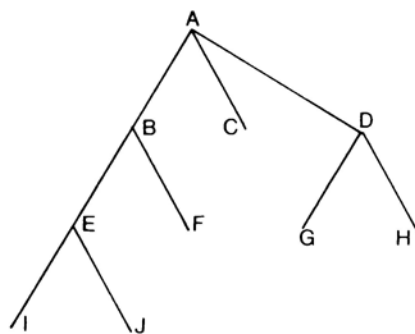
La conoscenza degli alberi binari è importante perchè un qualunque albero ordinato *A* può venir rappresentato da un albero binario (*A*,*B*).

Le regole per passare da una forma all'altra sono le seguenti:

- esiste una corrispondenza biunivoca fra i nodi di *A* e di (*A*,*B*);
- il primo figlio di un nodo di *A* è il susseguente di sinistra del nodo corrispondente in (*A*,*B*);
- gli altri figli di un nodo di *A* sono i susseguenti di destra di questo stesso nodo in (*A*,*B*), e formano una catena. I collegamenti diretti tra figli e padre sono soppressi.

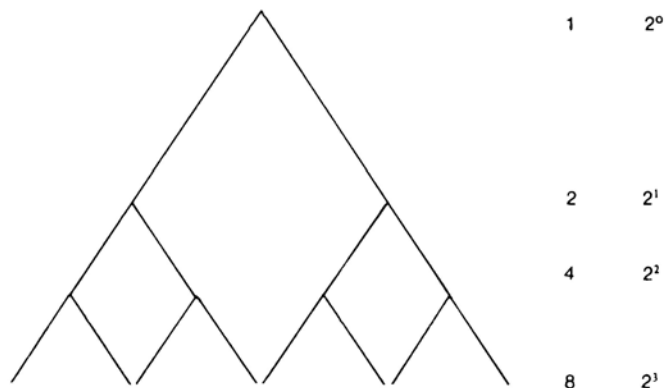
In tal modo l'albero *A* diventa

4.3 - Accesso ed inserzione nelle strutture ad albero

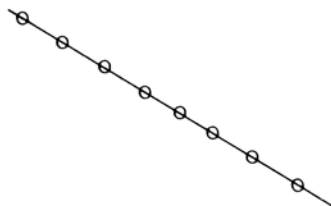
Il problema della ricerca di un elemento, e quello dell'inserzione e della cancellazione di un dato sono più complessi in una struttura ad albero che in una struttura lineare.

Allora, conoscendo la struttura dell'albero, possiamo utilizzare il nome di un percorso per accedere ad un elemento:

Il nome del percorso *A.B.E.I.* servirà ad accedere all'elemento *I*.

Se non si conosce a priori la struttura, per avere l'elemento desiderato bisognerà effettuare un attraversamento.

In pratica, molto spesso è nota la struttura ad albero dei livelli superiori, e questo permette di cercare l'elemento desiderato in un sottoalbero ad accesso rapido.

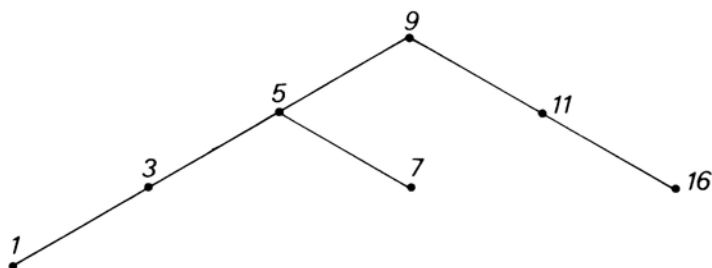
Ad esempio, con un albero binario ad *n* livelli si può accedere a 2^n elementi contenuti nelle foglie dell'albero.

Si può dimostrare che il tempo di ordinamento per un albero binario è dell'ordine di *n log n*.

Se la struttura è equilibrata, in media si accede ad un elemento terminale in *n* confronti: si ha quindi un accesso in *log n*. Ma, in pratica, la struttura ad albero non è sempre equilibrata; potremo ad esempio averne una come questa:

In questo caso l'accesso ad un elemento richiede lo stesso tempo che occorre con una lista lineare: pertanto bisogna cercare di avere alberi il più possibile equilibrati.

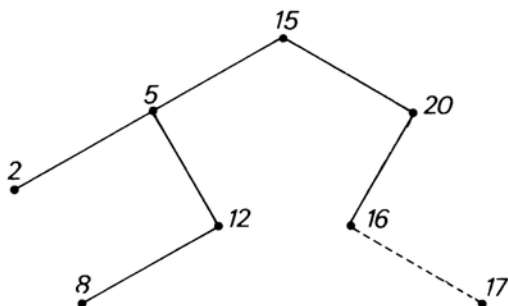
4.4 - Gli alberi equilibrati (alberi A.V.L.)

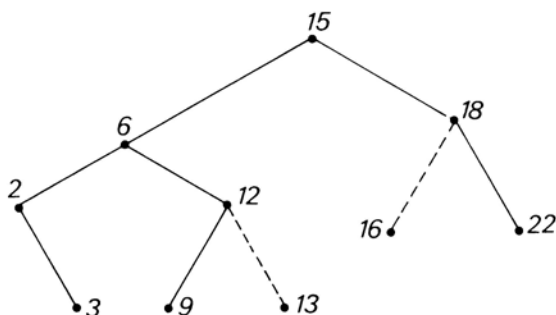
Il metodo di equilibratura degli alberi che esponiamo qui è stato proposto da Adel'son - Vel'ski e Landis nel 1962, donde il nome di alberi A.V.L.

La definizione è la seguente: per ciascun nodo di un albero A.V.L. la lunghezza del percorso più lungo del sottoalbero di sinistra si differenzia da quella del sottoalbero di destra di un'unità al massimo.

Esempio

Quando viene inserito, o cancellato, un elemento, allora si pone il problema della conservazione della proprietà A.V.L.

Supponiamo che nelle foglie dell'albero venga inserito un nuovo elemento. L'inserzione può squilibrare l'albero, che in tal modo non sarà più A.V.L.

Esempio

Qui l'albero non è più A.V.L. perchè il sottoalbero sinistro del nodo 20 ha lunghezza 2, mentre il sottoalbero destro ha lunghezza 0.

Ma esistono dei casi in cui la proprietà A.V.L. può essere conservata, come in questo esempio:

Quindi diremo che *un nodo è equilibrato* se i percorsi più lunghi dei due sottoalberi che partono da esso sono uguali; se il percorso del sottoalbero di sinistra è più lungo di un'unità rispetto a quello del sottoalbero di destra, diremo che il nodo è *pesante a sinistra*; se poi il nodo non è nè equilibrato nè pesante a sinistra, sarà *pesante a destra*. In un albero A.V.L. i nodi rientrano in una di queste condizioni.

4.4.1 - Aggiunta di un elemento ad un albero A.V.L.

Quest'operazione fa sì che uno o più nodi cambino condizione.
Sono possibili tre casi:

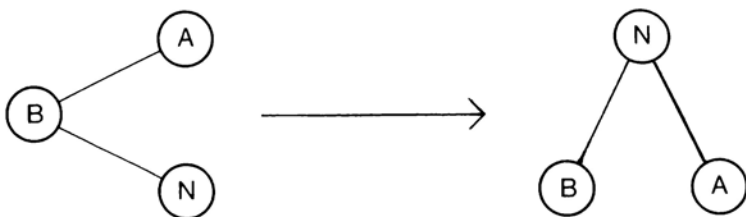
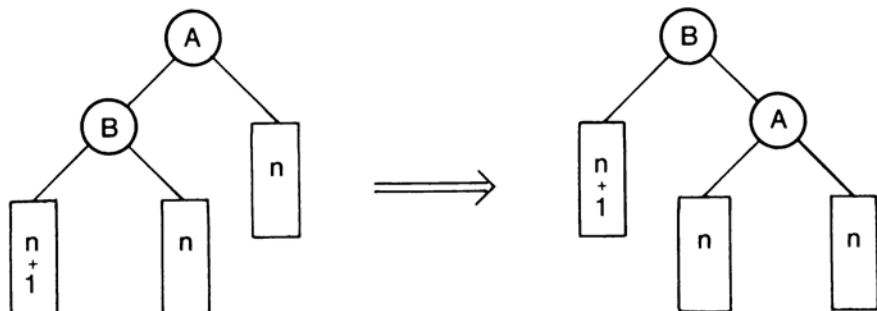
1) Un nodo equilibrato diventa pesante a sinistra o pesante a destra.

2) Un nodo pesante (a sinistra o a destra) diventa equilibrato.

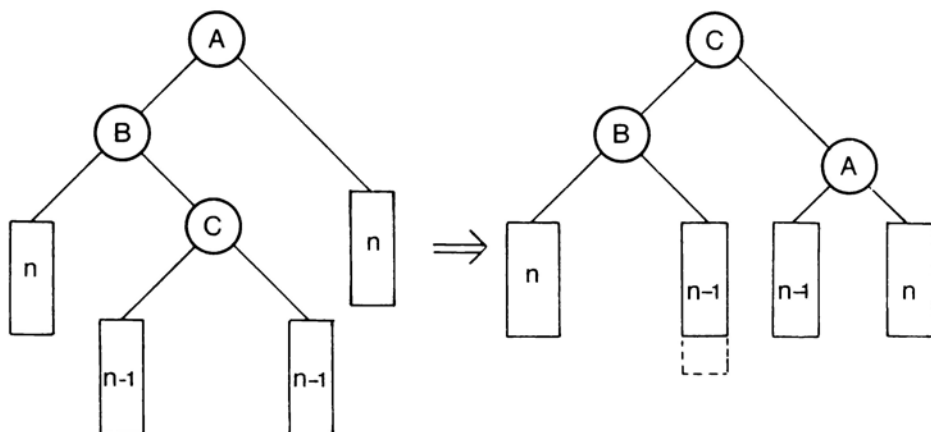
3) Un nodo pesante (a sinistra o a destra) diventa totalmente squilibrato per l'inserzione di un nuovo elemento nel sottoalbero già precedentemente pesante. Si dice allora che il nodo diventa *critico*.

I primi due casi non inficiano la proprietà A.V.L. dell'albero; infatti nel caso 1) il nodo precedente cambia condizione, diventando pesante (a sinistra o a destra), mentre nel caso 2) il nodo precedente non cambia condizione, perchè la lunghezza del percorso più lungo del sottoalbero resta invariata.

Nel caso 3), invece, l'albero, per conservare la proprietà A.V.L., dev'essere riorganizzato: questo può essere fatto in uno dei modi seguenti:

a) L'albero relativo al nodo critico dopo l'inserimento ha tre soli elementi: allora, per riequilibrare l'albero, è sufficiente porre *N* come radice del sottoalbero.

b) Il secondo caso, più complesso, è rappresentabile così:

Alla sinistra di *B* è stato aggiunto un elemento, e questo squilibra il sottoalbero sinistro di *A*.

Per realizzare una struttura equilibrata, basta effettuare una rotazione verso destra che ponga *B* come radice. Lo stesso può accadere a destra.

c) Il terzo caso è rappresentato da

Se viene aggiunto un elemento ad un sottoalbero di *C*, il nodo *A* diventa pesante a sinistra.
Allora, per riequilibrare *A*, è sufficiente effettuare una doppia rotazione, prima (*B*,*C*) verso sinistra, poi (*C*.*A*) verso destra.

Ed ecco un programma che riporta in equilibrio un albero A.V.L.

```
PROGRAM alberoavl;
TYPE
  puntatore = ↑nodo;
  condizione = (leggero, equil, pesante);
  nodo = RECORD nome: string;
                sinistra: puntatore;
                destra: puntatore;
                avl: condizione
         END;
VAR
  radice: puntatore;
  parola: string;
  b: boolean;
PROCEDURE scriveralb (p: puntatore);
BEGIN
  IF p < > NIL THEN
    BEGIN
      scriveralb (p↑.sinistra);
      writeln (p↑.nome);
      scriveralb (p↑.destra);
    END;
END;
PROCEDURE inserire (VAR p: puntatore; m: string; nn: boolean);
VAR
  ps, pd: puntatore;
BEGIN
  IF p = NIL THEN
  (*nuovo nodo*)
    BEGIN
      nn: = true;
      new (p);
      WITH p↑ DO
        BEGIN
          nome: = m;
          sinistra: = nil;
          destra: = nil;
```

```
          avl: = equil;
        END;
    END
  ELSE
    IF m > p↑.nome THEN
      (*inserzione a sinistra*)
      BEGIN
        inserire (p↑.sinistra, m, nn);
        IF nn THEN
          (*ramo di sinistra equilibrato?*)
            CASE p↑.avl OF
              pesante: BEGIN
                         nn: = false;
                         p↑.avl: = equil;
                       END;
              equil: p↑.avl: = leggero;
              leggero: BEGIN
                         ps: = p↑.sinistra;
                         IF ps↑.avl = leggero THEN
                           BEGIN
                             (*rotazione ss*)
                             p↑.sinistra: = ps↑.destra;
                             ps↑.destra: = p;
                             p↑.avl: = equil;
                             p: = ps;
                           END;
                         ELSE
                           BEGIN
                           (*doppia rotazione*)
                             pd: = ps↑.destra;
                             ps↑.destra: = pd↑.sinistra;
                             pd↑.sinistra: = ps;
                             p↑.sinistra: = pd↑.destra;
                             pd↑.destra: = p;
                             IF pd↑.avl = leggero THEN
                               p↑.avl: = pesante
                               ELSE p↑.avl: = equil;
                             IF pd↑.avl = pesante THEN
                               ps↑.avl: = leggero
                               ELSE ps↑.avl: = equil;
                             p: = pd;
                           END;
                         nn: = false;
```

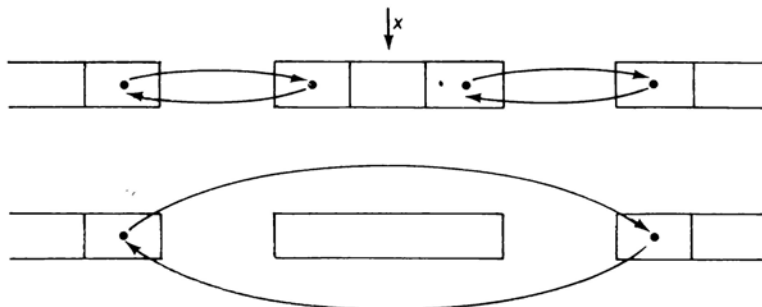
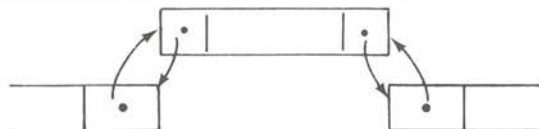
```
                    p↑.avl: = equil;
                  END; (*caso leggero*)
        END; (*fine del caso*)
    END
  ELSE
    IF m > p↑.nome THEN
      (*inserire a destra*)
      BEGIN
        inserire (p↑.destra, m, nn);
        IF nn THEN
          CASE p↑.avl OF
            leggero:  BEGIN
                        nn: = false;
                        p↑.avl: equil;
                      END;
            equil: p↑.avl: = pesante;
            pesante:  BEGIN (*riequilibrare*)
                        pd: = p↑.destra;
                          IF pd↑.avl = pesante THEN
                            BEGIN
                              p↑.destra: = pd↑.sinistra;
                              pd↑.sinistra: = p;
                              p↑.avl: = equil;
                              p: = pd;
                            END
                          ELSE
                            BEGIN
                            (*doppia rotazione destra sinistra*)
                              ps: = pd↑.sinistra;
                              pd↑.sinistra: = ps↑.destra;
                              ps↑.destra: = pd;
                              p↑.destra: = ps↑.sinistra;
                              ps↑.sinistra: = p;
                              IF ps↑.avl: = pesante THEN
                                p↑.avl: = leggero
                                ELSE p↑.avl: = equil;
                              IF ps↑.avl = leggero THEN
                                pd↑.avl: = pesante
                                ELSE pd↑.avl: = equil;
                              p: = ps;
                            END;
                          nn: = false;
                          p↑.avl: = equil;
```

```
                    END; (*caso pesante a sinistra*)
               END; (*fine del caso*)
          END
        ELSE nn: = false;
END; (*fine inserire*)
(*programma principale*)
BEGIN
  radice: = nil;
  WHILE length (parola) < > 0 DO
    BEGIN
      write ('parola:');
      readln (parola);
      inserire (radice, parola, b);
      writeln;
      writeln ('albero avl:');
      scriveralb (radice);
      writeln;
      END;
END.
```

ESERCIZI

1. Scrivere i programmi che permettono di accedere ad un elemento qualsiasi di una lista concatenata.

2. Scrivere i programmi che permettono di realizzare la fusione e la divisione di due liste concatenate.

3. Inserimento e cancellazione di un elemento in una lista a doppio senso.
   La cancellazione si può schematizzare come segue:

L'inserimento è più complesso:

Scrivere un programma che realizzi queste operazioni.

CAPITOLO 8

IL TRATTAMENTO GRAFICO IN PASCAL

"Non dallo spazio devo ricercare la mia dignità, ma dal modo in cui è regolato il mio pensiero. Se possedessi delle terre, non per questo avrei di più: con lo spazio, l'Universo mi comprende e m'inghiotte; con il pensiero, io lo comprendo".

PASCAL,
Pensées

In questo capitolo vedremo come si possono risolvere dei problemi grafici in Pascal. Pur non rientrando tali problemi nel linguaggio standard, molti sistemi propongono delle istruzioni che permettono di trattarli. Solo pochi anni fa queste operazioni richiedevano l'acquisto di terminali grafici, relativamente costosi; attualmente invece sono realizzabili con un qualunque sistema collegato ad un normale televisore: in particolare, i programmi che presentiamo ora sono stati eseguiti su un sistema Apple, che permette anche l'uso del colore.

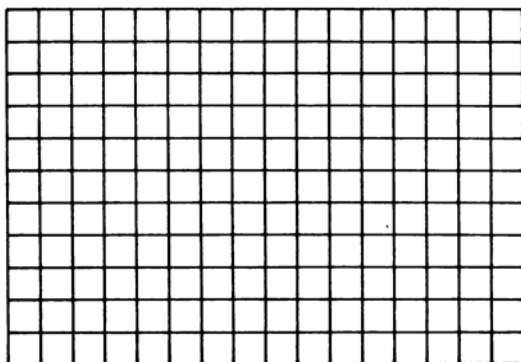
Vedremo che le capacità grafiche conferiscono al linguaggio un'altra dimensione; si può dire quindi che è un peccato che non siano meglio standardizzate, perchè, dal punto di vista pedagogico, è spesso un metodo più facile e divertente spiegare i principi della programmazione con l'aiuto di programmi visivi.

Al di là delle caratteristiche particolari del calcolatore utilizzato, insisteremo sui problemi generali posti dalla grafica: il cambiamento di origine, il cambiamento di scala, il trattamento dei vettori. Così quanti non dispongono di un Apple potranno applicare queste informazioni ad un sistema diverso.

All'inizio del capitolo, parleremo poi di una modalità semigrafica che si limita a realizzare la visualizzazione di punti mediante le procedure di uscita dei caratteri standard.

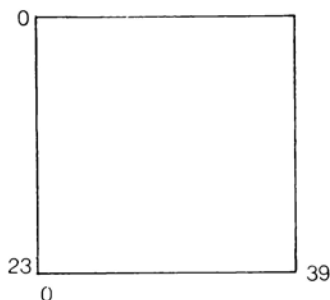
1 - NOTE INTRODUTTIVE SULLA GRAFICA IN PASCAL

Il linguaggio Pascal standard non prevede istruzioni grafiche, ma in ogni caso diversi microcalcolatori le forniscono: ad esempio il sistema Apple, che prenderemo come esempio in questo capitolo.

L'Apple, utilizzando un comune televisore, è il sistema grafico meno costoso; dispone inoltre delle procedure grafiche del sistema dell'U.C.S.D. La *modalità grafica* prevede che lo schermo sia scomposto in una griglia di punti che vengono indirizzati con un sistema di coordinate cartesiane: ascissa (x) e ordinata (y).

L'impiego dello schermo come griglia di punti (che sono poi in realtà dei quadratini) è specificato mediante un comando particolare.

Esistono due modalità grafiche, con una delle quali non è necessario lo schermo grafico: la prima, che chiameremo *grafica semplice*, usa una griglia 24 x 40 nella modalità carattere usuale (24 righe di 40 caratteri); la seconda è la modalità grafica *ad alta risoluzione*, ed usa una griglia 279 x 191, corrispondente all'intero schermo.

2 - LA MODALITÀ GRAFICA SEMPLICE

È usata una griglia 24 x 40; ciascun punto è individuato da coordinate che vanno da 0 a 39 per x e da 0 a 23 per y; il punto di coordinate (0,0) è quello posto nell'angolo in alto a sinistra dello schermo.

Le ascisse aumentano andando da sinistra verso destra, mentre le ordinate aumentano andando dall'alto verso il basso, contrariamente a quanto accade tipicamente in matematica.

In questa modalità esistono solo il bianco e il nero.

2.1 - Visualizzazione di un punto

Quest'operazione richiede il ricorso alla procedura standard di uscita carattere sullo schermo. Il movimento del cursore permette di posizionarsi su un punto qualsiasi dello schermo.

In modalità testo esiste generalmente una procedura che consente di spostare il cursore in un punto dello schermo di coordinate *x*, *y*, corrispondente alla posizione di un carattere sullo schermo.

Nel sistema U.C.S.D. questa procedura è la *gotoxy* (*x*, *y*), ove

$0 \leqslant x \leqslant 39$
$0 \leqslant y \leqslant 23$

In particolare, questa procedura posiziona automaticamente il cursore, allo scopo di effettuare delle operazioni interattive.

Per visualizzare un punto si richiede la procedura

```
gotoxy (x, y)
```

seguita da un'istruzione che scrive un punto:

```
write ('.')
```

Esempio

Per visualizzare un punto di coordinate 20, 20 scriveremo:

```
gotoxy (20, 20);
write ('.');
```

Volendo disporre dell'intero schermo al fine di visualizzare un grafico, richiederemo all'inizio la procedura *page* (*output*), che cancella tutto lo schermo.

2.2 - Spostamento di un punto sullo schermo

Finora abbiamo esaminato solo una visualizzazione di tipo statico; ma in alcuni casi può essere utile far muovere un punto sullo schermo. Per far questo bisogna cambiare la posizione del punto: se si vuol visualizzare la traiettoria del punto, non c'è altro da fare; se invece non si vuole visualizzare la traiettoria, ma solo la posizione oc-

cupata da un punto in funzione del tempo, bisogna cancellare la vecchia posizione. Questo è il principio dell'animazione.

Esempio

Scriviamo un programma che sposti il cursore sullo schermo e lo faccia "rimbalzare" sui bordi dello schermo:

```
PROGRAM cursore;
uses Applestuff;
CONST
  riga = 23;
  colo = 39;
VAR
  x: 0 ... colo;
  y: 0... riga;
  dx, dy: -1... +1;
BEGIN
  x: = 0; y: = 0:
  dx: = 1; dy: = 1;
  page (output);
  WHILE true DO
    BEGIN
      y: = y + dy;
      x: = x + dx;
      IF x = 0 THEN dx: = 1;
      IF y = 0 THEN dy: = 1;
      IF x = colo THEN dx: = -1;
      IF y = riga THEN dy: = -1;
      gotoxy (x, y);
    END;
END.
```

Il movimento del cursore si ottiene senza ricorrere a procedure di scrittura.

2.3 - Tracciamento per punti di un segmento di retta qualsiasi

Si abbia la retta espressa dall'equazione

$$y = ax + b$$

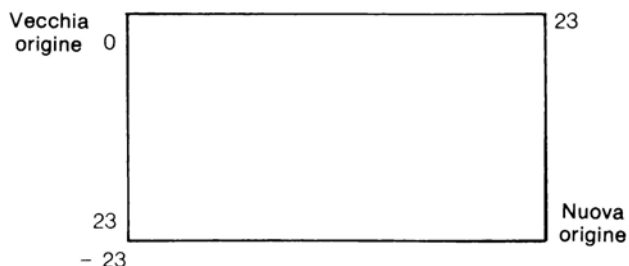
In modalità grafica semplice nessuna istruzione permette di tracciare una retta, perchè la griglia di punti è troppo poco fitta, e quindi i tratti che si ottengono sono delle sequenze di punti che si approssimano ad una retta.

Nel programma che segue supporremo che l'origine di *y* si trovi in basso a sinistra, per cui calcoleremo:

y = 23 − parte intera (a · x + b)

Il programma è questo:

```
PROGRAM retta;
CONST
  riga = 23;
  colo = 39;
VAR
  x: 0 ... colo;
  y: integer;
  dx, dy: −1... +1;
  a, b: real;
BEGIN
  read (a, b);
  FOR x: = 0 TO colo DO
    BEGIN
      y: = riga − round(a* x + b);
      IF (y > = 0) OR (y < = riga) THEN
        BEGIN
          gotoxy (x, y);
          write ('.');
        END;
    END;
END.
```

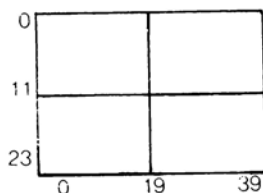
Attenzione: letti i parametri *a* e *b*, può capitare che la retta si trovi fuori dello schermo. Ad esempio, questo programma non consente di tracciare una retta espressa dall'equazione $y = -x - 1$ perchè, per avere dei valori di *y* positivi, occorre attribuire ad *x* dei valori negativi. In un caso come questo bisogna effettuare un cambiamento d'origine, ponendola ad esempio in basso, all'ascissa 23.

Così facendo, verrà tracciata una retta espressa dall'equazione

$$23 - y = (x - 23) - 1 = - x + 23 - 1$$

per cui, con la nuova origine, $y = x + 1$.

2.4 - Il tracciamento di curve in modalità grafica semplice

Anche qui, la dimensione della griglia consente di ottenere, nel migliore dei casi, delle curve approssimate.

Per avere la garanzia di ottenere tutte le porzioni delle curve poste nei quattro quadranti, bisogna effettuare dei cambiamenti degli assi: ad esempio assumeremo che il punto centrale, di ascissa 19 e ordinata 11, sia l'origine degli assi. Avremo cosi un'idea dell'andamento della curva, in base alla quale cambiare più opportunamente l'origine.

2.4.1 - Cambiamento d'origine

Se la nuova origine ha coordinate x_0, y_0, sappiamo che la nuova equazione di una curva $y = f(x)$ sarà

$$y - y_0 = f(x - x_0)$$

e, invertendo il senso di y, avremo:

$$-y + y_0 = f(x - x_0)$$

Esempio

Per $x_0 = 0$ e $y_0 = 23$, avremo:

$$y = 23 - f(x)$$

2.4.2 - Il tracciamento degli assi

Vogliamo scrivere una procedura che permetta di tracciare gli assi di un sistema di assi cartesiani. I parametri di questa procedura sono le coordinate x_0 ed y_0 dell'origine.

L'asse delle x si ottiene con caratteri "–", l'asse delle y con caratteri "I". Ed ecco il programma retta modificato:

```
PROGRAM retta;
CONST
  riga = 23;
  colo = 39;
VAR
  nx, x0, x: integer;
  y0, y: integer;
  a, b: real;
PROCEDURE assi (x0, y0: integer);
BEGIN
  FOR x: = 0 TO colo DO
    BEGIN
      gotoxy (x, y0);
      write ('—');
    END;
  FOR y: = 0 TO riga DO
    BEGIN
      gotoxy (x0, y);
      write ('I');
    END;
END;
BEGIN
  write ('origine'); readln (x0, y0);
  write ('parametro='); read (a, b);
  page (output);
  assi (x0, y0);
  FOR x: = 0 TO colo DO
    BEGIN
      nx: = x - x0;
      y: = y0 - round(a * nx + b);
      IF (y > = 0) AND (y < = riga) THEN
        BEGIN
          gotoxy (x, y);
          write ('.');
        END;
    END;
END.
```

2.4.3 - Il tracciamento di curve

Il programma precedente può essere facilmente adattato per tracciare una curva per punti: basterà scrivere una funzione che calcoli $y = f(x)$, e sostituire l'istruzione che dà la coordinata di y con

$$y := y_0 - f(x, p_1, p_2, \dots p_n)$$

ove le p_i sono parametri.
Ed ecco un esempio di tracciamento di una curva parabolica:

```
PROGRAM curva;
CONST
  riga = 23;
  colo = 39;
VAR
  nx, x0, x: integer;
  y0, y: integer;
  a, b, c: real;
PROCEDURE assi (x0, y0: integer);
BEGIN
  FOR x: = 0 TO colo DO
    BEGIN
      gotoxy (x, y0);
      write ('—');
    END;
  FOR y: = 0 TO riga DO
    BEGIN
      gotoxy (x0, y);
      write ('I');
    END;
END;
FUNCTION parabola(a, b, c: real; x: integer): integer;
BEGIN
  parabola: = round(a * x * x + b * x + c);
END;
BEGIN
  write ('origine'); readln (x0, y0);
  write ('parametri='); read (a, b, c);
  page (output);
  assi (x0, y0);
  FOR x: = 0 TO colo DO
    BEGIN
      nx: = x — x0;
      y: = y0 — parabola (a, b, c, nx);
      IF (y > = 0) AND (y < = riga) THEN
        BEGIN
          gotoxy (x, y);
          write ('.');
        END;
    END;
END.
```

2.4.4 - Rappresentazione di un istogramma

Supponiamo di voler rappresentare sotto forma d'istogramma le caratteristiche di un gruppo d'individui divise in classi, in modo che la classe più numerosa sfrutti tutta l'altezza disponibile sullo schermo.

Il programma che segue è costituito da una funzione *scala*, che calcola la scala opportuna per rappresentare le classi, e tre procedure.

La prima procedura, *barra*, scrive una barra, di altezza nota, dell'istogramma, per mezzo di asterischi.

La seconda procedura, *altezza*, calcola l'altezza delle classi.

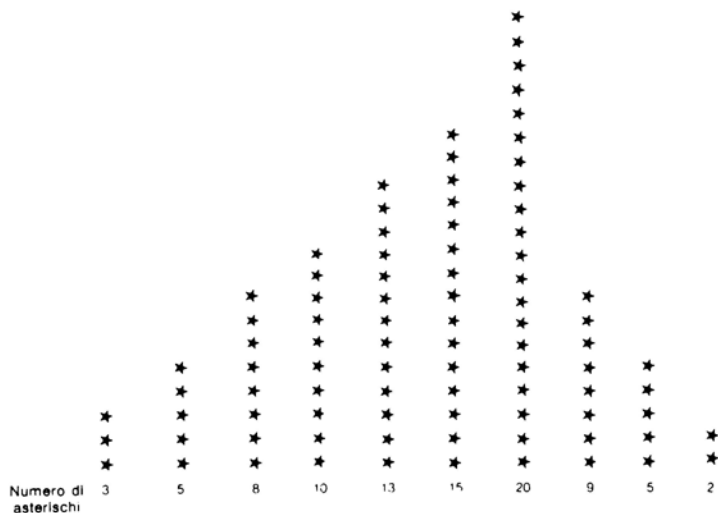
La terza, cioè la procedura *istogramma*, traccia l'istogramma a partire da un'origine fissata.

Il programma è il seguente:

```
PROGRAM isto;
CONST
  riga = 23;
  colo = 39;
VAR
  n, x0, x: integer;'
  i, y0, y: integer;
  a: ARRAY [0 ... colo] OF real;
  n: ARRAY [0... colo] OF integer;
FUNCTION scala(y0, n: integer): real;
VAR
  max: real;
BEGIN
  max: = a[1];
  FOR i: = 2 TO n DO
    IF a[i] > a[i −1] THEN
      max: = a[i];
  scala: = y0/max;
END;
PROCEDURE barra(y0, x, k: integer);
BEGIN
  FOR y: = y0 DOWNTO y0 − k + 1 DO
    BEGIN
      gotoxy (x, y);
      write ('*');
    END;
END;
PROCEDURE altezza (y0, n: integer);
VAR
  sca: real;
```

```
BEGIN
   sca: = scala (y0, n);
   FOR i: = 1 TO n DO
      n[i]: = round(a[i] * sca);
END;
PROCEDURE istogramma (x0, y0, n: integer);
BEGIN
   altezza (y0, n);
   FOR x: x0 TO x0 + n - 1 DO
      barra (y0, x, n[x - x0 + 1]);
END;
BEGIN
   write ('origine'); readln (x0, y0);
   write ('nr di classi='); readln (n);
   write ('nr elementi nelle classi=');
   FOR i: = 1 TO n DO
      READ (a[i]);
   page (output);
   istogramma (x0, y0, n);
END.
```

Esecuzione

```
origine 10 20 (R)
nr di classi = 10 (R)
nr elementi nelle classi = 5 10 15 20 25 30 40 18 10 4 (R)
```

3 - LA GRAFICA AD ALTA RISOLUZIONE

Abbiamo visto che la grafica a bassa risoluzione presenta dei limiti quando si tratta di tracciare delle curve. La grafica ad alta risoluzione utilizza una griglia di punti molto più piccoli, e di conseguenza permette di ottenere dei tracciati molto più precisi.

Ricordiamo che sul sistema Apple la grafica ad alta risoluzione utilizza una griglia di punti 279 x 191 estesa a tutto lo schermo.

La modalità grafica presente sul sistema U.C.S.D. è particolare, in quanto utilizza procedure per lo spostamento di un punto detto del moto "a tartaruga".

3.1 - La grafica della "tartaruga"

È una modalità grafica particolare, sviluppata da S. Papert ed il suo gruppo di ricerca presso il M.I.T. (Massachusetts Institute of Technology). Il suo scopo è quello di consentire ai bambini, e alle persone che ignorano il principio delle coordinate cartesiane, di descrivere algoritmi di tracciati grafici. Il principio di base è quello di far spostare un punto, individuato da una "tartaruga" capace di avanzare di un certo numero di passi e di girare di un certo angolo: si fa in qualche modo riferimento a delle coordinate "polari relative". Questo metodo trova applicazione nel sistema Pascal U.C.S.D., e nel sistema Apple in particolare.

La grafica della "tartaruga" è costituita da una biblioteca di procedure che vengono richieste mediante il comando

```
uses turtlegraphics
```

3.1.1 - Inizializzazione

L'inizializzazione della modalità grafica avviene mediante la procedura *initturtle* (cioè inizializzazione della tartaruga), che produce questi effetti:
- si passa alla modalità grafica;
- si pulisce lo schermo;
- si colloca la tartaruga al centro dello schermo.

Questo comando presuppone che l'intero schermo venga usato in modalità grafica, e che non venga definito il colore. Il colore di fondo dello schermo è generalmente il nero.

3.1.2 - Passaggio dalla modalità grafica alla modalità testo, e viceversa

Se in un programma si vuol utilizzare lo schermo per visualizzare sia un testo che un disegno, si ricorrerà alternativamente alla procedura *grafmode* (modalità grafica) ed alla procedura *textmode* (modalità testo).

La procedura *grafmode*, diversamente da *initturtle*, non determina l'inizializzazione, ma semplicemente visualizza la "finestra" grafica del programma.

Analogamente, la procedura *textmode* permette di tornare in modalità testo, conservando tuttavia in memoria il contenuto della finestra grafica già utilizzata.

Al termine di un programma, per riprendere il controllo in modalità testo, si ricorre generalmente alla procedura *textmode*.

3.1.3 - Il colore

Se si vuole che la visualizzazione sullo schermo sia a colori, si ricorrerà al comando *pencolor* (colore della penna).

I colori cambiano a seconda dei sistemi. Sul sistema Apple, ad esempio, si hanno questi colori: bianco (*white*), nero (*black*), verde (*green*), viola (*violet*), arancione (*orange*), azzurro (*blue*).

Ad esempio, con *pencolor* (*orange*) si definisce il colore arancione.

Con uno schermo in bianco e nero, si hanno di fatto solo questi due colori. Comunque sui sistemi Apple esistono diverse versioni dei colori bianco e nero, a seconda che vengano o no usati insieme con altri colori. Infatti i colori diversi dal bianco e dal nero sono determinati da tratti grafici più spessi: generalmente, per avere un buon effetto visivo, sono necessari due punti. Esistono tre varietà di bianco e di nero: *white* e *black* per i tratti sottili, *white1* e *black1* per il bianco e il nero usati con il verde o il viola, *white2* e *black2* per il bianco e il nero usati con l'arancione e l'azzurro.

Beninteso, queste caratteristiche sono specifiche dell'Apple; le abbiamo esposte unicamente per facilitare la comprensione dei programmi che compaiono nel capitolo.

Nel parametro della procedura *pencolor* si può indicare l'assenza di colore scrivendo *pencolor* (*none*), che non modifica la traccia dell'immagine già visualizzata.

Inoltre, con la visualizzazione bianco su nero, si può richiedere l'inversione del fondo (negativo), per cui il nero diventa bianco, e viceversa. Quest'operazione è effettuata per mezzo del parametro *reverse*: *pencolor* (*reverse*).

3.1.4 - La finestra grafica

In assenza di altre indicazioni, la modalità grafica utilizzerà tutto lo schermo disponibile. Si può comunque definire una finestra grafica per poter lavorare solo su una parte dello schermo; per far questo occorre specificare le coordinate dei limiti destro, sinistro, superiore ed inferiore della finestra, mediante la procedura *viewport*, i cui parametri sono, nell'ordine, sinistra, destra, basso, alto. I parametri devono avere dei valori corrispondenti ai limiti della finestra che si vuole ottenere, cioè

$0 \leqslant$ sinistra $\leqslant 279$
$0 \leqslant$ destra $\leqslant 279$
$0 \leqslant$ basso $\leqslant 191$
$0 \leqslant$ alto $\leqslant 191$

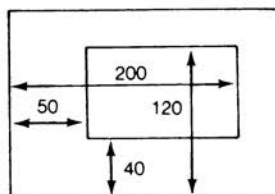
con

sinistra < destra

e

basso < alto

Esempio

viewport (50, 200, 40, 120) definirà la finestra rappresentata nella figura seguente:

Inoltre, in questo sistema Pascal, il taglio automatico del disegno permette che i limiti della finestra siano superati senza che per questo si abbia un messaggio di errore: i punti che cadono fuori della finestra vengono ignorati, per cui si possono tracciare i disegni senza preoccuparsi del problema dei limiti dello schermo.

Può anche capitare, qualora il fattore di scala sia tale per cui tutti i punti si trovino al di fuori dello schermo, che non venga visualizzato nessun punto.

3.1.5 - Riempimento e cancellazione dello schermo grafico

Si può creare per lo schermo uno sfondo di un determinato colore mediante la procedura *fillscreen* (riempimento dello schermo), in cui il parametro *penmode* definisce il colore. Ad esempio, *fillscreen* (*green*) riempie la finestra definita nello schermo con il verde.

Per avere il fondo nero, in altre parole per *cancellare* tutto lo schermo, scriveremo *fillscreen* (*black*).

Il programma che segue effettua una verifica delle procedure elementari *viewport* e *fillscreen*.

```
PROGRAM grafica;
uses turtlegraphics;
VAR
  colore: screencolor;
BEGIN
  initturtle; viewport (50, 200, 40, 120);
  FOR colore: = none TO white2 DO
    fillscreen (colore);
  textmode
END.
```

3.2 - Le procedure grafiche di movimento della tartaruga

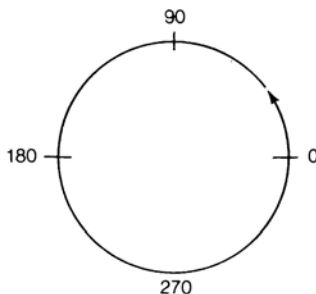
Le procedure di movimento della tartaruga permettono di spostarsi in una data direzione e di cambiare direzione: quando il parametro di *pencolor* è diverso da *none* o dal colore di fondo, sullo schermo appare la traccia di questi movimenti.

3.2.1 - Le procedure di cambiamento di direzione

Esistono due procedure che fanno voltare la tartaruga, delle quali l'una utilizza un parametro angolare relativo, l'altra un parametro angolare assoluto: sono, rispettivamente, le procedure *turn* (angolo) e *turnto* (angolo).

Il parametro angolare è espresso sotto forma di un intero modulo 360, e rappresenta quindi un angolo espresso in gradi.

L'angolo zero corrisponde alla posizione orizzontale, con direzione verso destra.

Gli angoli sono definiti sul cerchio trigonometrico, quindi in senso inverso rispetto al senso delle lancette dell'orologio.

La procedura *turnto* permette di posizionarsi nella direzione definita dal parametro angolare; la procedura *turn* fa incrementare l'angolo relativo alla direzione attuale della tartaruga dell'angolo espresso dal parametro angolare. Queste due procedure *non modificano* l'immagine sullo schermo: infatti, fino a quando la tartaruga resta ferma, non hanno alcun effetto visibile.

3.2.2 - Le procedure di movimento della tartaruga

Sono due: *move* e *moveto*, delle quali soltanto la prima si riferisce al movimento della tartaruga nella direzione attuale; la procedura *moveto* infatti autorizza lo spostamento in un punto di cui siano note le coordinate cartesiane.

Vediamo prima la procedura *move*.

Il suo unico parametro, intero, indica la lunghezza dello spostamento da effettuare:

```
move (distanza)
```

Move fa avanzare la tartaruga di una distanza stabilita dal paramero, *nella direzione attuale* della tartaruga.

Esempi

1)
```
PROGRAM quadrato;
uses turtlegraphics;
VAR i: integer;
BEGIN
  initturtle;
  pencolor (green);
  FOR i: = 1 TO 4 DO
    BEGIN
      move (60);
      turn (90);
    END;
END.
```

Questo programma permette di disegnare un quadrato con lato di lunghezza 60.

2)
```
PROGRAM tartaruga;
uses turtlegraphics;
VAR i: integer;
BEGIN
  initturtle;
  pencolor (violet);
  FOR i: = 0 TO 360 DO
    BEGIN
      move (8);
      turn (i);
    END;
END.
```

Questo programma invece permette di disegnare una spirale.

La procedura *moveto* differisce dalla precedente per il fatto che presuppone un riferimento a delle coordinate cartesiane: i suoi argomenti sono due interi che danno le coordinate *x* ed *y* di un punto.

Per un sistema Apple, ad esempio, si ha:

$0 \leqslant x \leqslant 279$
$0 \leqslant y \leqslant 191$

Questi valori variano a seconda della risoluzione dello schermo con il quale si lavora.

La chiamata di questa procedura è

```
moveto (x, y)
```

Il suo effetto è di spostare la tartaruga in linea retta dal punto in cui si trova al punto di coordinate x, y: il colore è specificato dalla procedura *pencolor*. Ad ogni modo la direzione iniziale della tartaruga *non viene modificata*.

All'inizio, a seguito dell'esecuzione della procedura *initturtle*, la tartaruga viene spostata al centro dello schermo senza essere visualizzata.

3.2.3 - Le funzioni di posizione della tartaruga

Per dare ad un programma la possibilità d'individuare il punto dello schermo in cui ci si trova, si può accedere alla posizione ed alla direzione attuale della tartaruga stessa. I parametri sono dati dalle seguenti funzioni intere:

a) *Posizione occupata dalla tartaruga*
Le funzioni *turtlex* e *turtley* danno la posizione delle coordinate cartesiane x ed y della tartaruga sullo schermo.

b) *Direzione della tartaruga*
La funzione è *turtleang*, che fornisce il valore dell'angolo attuale della tartaruga, compreso fra 0 e 359 gradi.

c) *Funzione di stato dello schermo*
Mediante la funzione booleana *screenbit* (x, y) si può conoscere lo stato di un punto dello schermo. x ed y sono le coordinate del punto sottoposto alla verifica, e il valore che si ottiene è *true* (vero) se il punto è visualizzato (se cioè non è nero); se invece è nero, il valore ottenuto è *false* (falso).

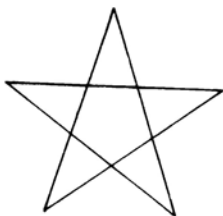
d) *Scrittura di caratteri sullo schermo in modalità grafica*
Con i sistemi che prevedono la modalità grafica della tartaruga, si possono scrivere dei caratteri sullo schermo.
La procedura *wchar* (*car*) permette di visualizzare un unico carattere nel punto in cui si trova la tartaruga: il suo parametro è un carattere.
Se la tartaruga, prima della richiesta della procedura, ha coordinate x ed y, dopo la sua esecuzione avrà coordinate $x + 7$, y.
Un'altra procedura disponibile è *wstring* (*stringa*), che permette di visualizzare una stringa di caratteri: il parametro è una stringa di caratteri.
Se la lunghezza della stringa è l, in seguito all'esecuzione della procedura la tartaruga sarà posizionata alle coordinate $(x + 7 \cdot l, y)$.

e) *Visualizzazione d'immagini grafiche*
Ricordiamo infine l'esistenza di procedure che permettono di visualizzare un insieme di punti sullo schermo, e di ottenere così delle immagini.
Queste procedure sono diverse da un sistema all'altro, per cui non ne presenteremo in dettaglio i parametri.

3.3 - Esempi di programmi grafici

3.3.1 - Stella a 2n + 1 punte

Per disegnare una stella a $2n + 1$ punte, con n che varia da 1 a 20, il programma è il seguente:

```
PROGRAM stella;
uses turtlegraphics;
VAR
   ang, n, d, i: integer;
BEGIN
   readln (n);
   ang: = 180 — 180 DIV n;
   read (d);
   initturtle;
   pencolor (violet);
   FOR i: = 0 TO n DO
      BEGIN
         move (d);
         turn (ang);
      END;
END.
```

3.3.2 - Spirale

Per disegnare una spirale occorre girare di un angolo costante, e spostarsi di una lunghezza che viene incrementata ad ogni cambiamento di posizione. Il programma sarà il seguente:

```
PROGRAM spirali;
uses turtlegraphics, Applestuff;
VAR
   delta, ang, n, d, i: integer;
   car: char;
BEGIN
   readln (n);
   ang: = 360 DIV n - 1;
   read (d);
   initturtle;
   pencolor (violet);
   delta: = 2;
      WHILE NOT keypress DO
         BEGIN
            move (d);
            turn (ang);
            d: = d + delta;
         END;
         textmode;
END.
```

3.3.3 - Tracciamento di un segmento di retta

Si abbia la retta espressa dall'equazione $y = ax + b$.

In modalità grafica ad alta risoluzione, il programma è quello che segue; per ottenere il tracciamento del segmento basta calcolarne le due estremità.

```
PROGRAM grafica;
uses turtlegraphics;
CONST
   xm = 279;
   ym = 191;
VAR
   a, b: real;
   y: integer;
BEGIN
   initturtle;
   WHILE a < > 0 DO
   BEGIN
      read (a, b);
      pencolor (none);
      moveto (0, round (b) );
      y: = round(a * xm + b);
```

```
      pencolor (green);
      moveto (xm, y);
    END;
    textmode;
  END.
```

Per tracciare un segmento che collega i punti (x_1, y_1) e (x_2, y_2) non conviene calcolare l'equazione della retta, cioè

$$\frac{x - x_1}{y - y_1} = \frac{x_2 - x_1}{y_2 - y_1}$$

Basta scrivere una procedura nel cui corpo compaia quanto segue:

```
BEGIN
  pencolor (none);
  moveto (x1, y1);
  pencolor (colore);
  moveto (x2, y2);
END;
```

Questa procedura è applicabile in particolare al tracciamento di assi ed alla visualizzazione di una finestra colorata.

Il programma seguente permette di spostare una finestra colorata sullo schermo: in esso compaiono tutte le procedure finora descritte.

```
PROGRAM grafica;
uses turtlegraphics;
CONST
  l = 279;
  a = 191;
VAR
  x0, y0, x1, y1, x2, y2: integer;
  dx, dy, x, y, j: integer;
PROCEDURE segmento (x1, y1, x2, y2: integer);
BEGIN
  pencolor (none);
  moveto (x1, y1);
  pencolor (violet);
  moveto (x2, y2);
END;
PROCEDURE assi (x0, y0: integer);
```

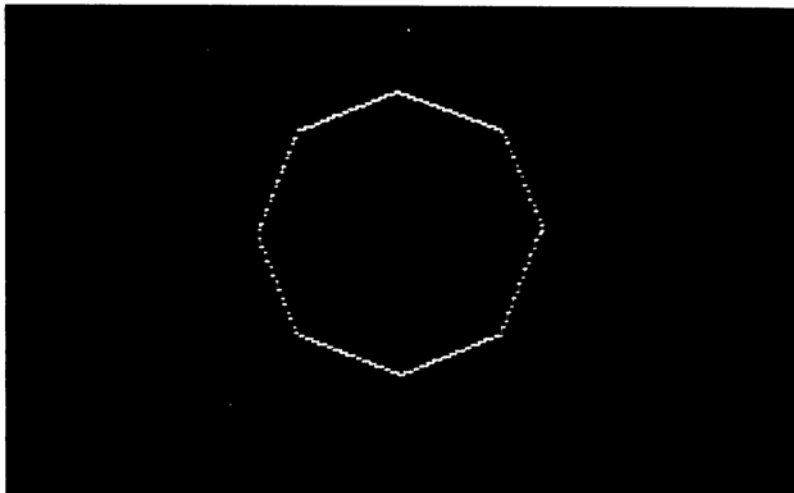
```
BEGIN
  segmento (0, y0, l, y0);
  segmento (x0, 0, x0, a);
END;
PROCEDURE finestra (x1, y1, j: integer);
VAR
  i: integer;
BEGIN
  FOR i: = 0 TO j DO
    segmento (x1, y1 + i, x1 + j, y1 + i);
END;
PROCEDURE spostafinestra (var x, y: integer;
  dx, dy, j: integer);
BEGIN
  x: = x + dx;
  y: = y + dy;
  finestra (x, y, j);
END;
BEGIN
  initturtle;
  assi (120, 80);
  x0: = 100, y0: = 100;
  finestra (x0, y0, 5);
  x: = x0; y: = y0; dx: = 5; dy: = 5;
  WHILE true DO
    BEGIN
      IF x > l THEN dx: = -dx;
      IF y > a THEN dy: = -dy;
      IF x < 0 THEN dx: = -dx;
      IF y < 0 THEN dy: = -dy;
    END;
END.
```

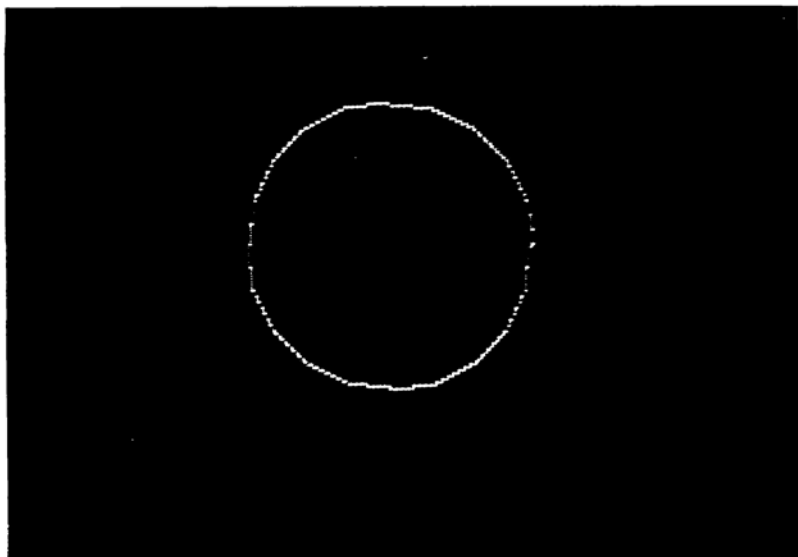
3.3.4 - Tracciamento di poligoni regolari

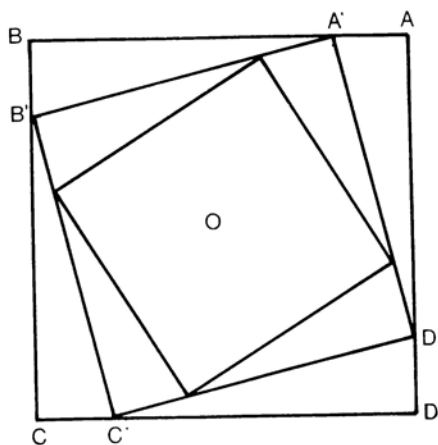
Si può inscrivere un poligono regolare in un cerchio, ed ottenere le coordinate dei vertici per mezzo di funzioni trigonometriche. Se il raggio del cerchio che include il poligono è *r*, le coordinate dei vertici saranno $r \cos \alpha$ ed $r \, sen \, \alpha$, dove α è l'angolo con vertice nel centro, sotto il quale è visto un lato del poligono: cioè $\alpha = 2\pi/n$, ove *n* è il numero dei lati.

La procedura che segue permette di disegnare un poligono di raggio e numero di lati qualsiasi.

Una volta calcolati l'angolo e la lunghezza del lato, il disegno sarà realizzato da un ciclo contenente la procedura *move* e la procedura *turn*.

```
PROCEDURE poligono (x0, y0, nrlato, raggio: integer);
VAR
  angolo: integer;
  i, lato: integer;
BEGIN
  angolo: = round(360/nrlato);
  lato: = round(2 * raggio * sin(angolo * pi/360));
  pencolor (none); moveto (x0 + raggio, y0);
  pencolor (green); turn (round ((180 + angolo)/2));
  FOR i: = 1 TO nrlato DO
    BEGIN
      move (lato);
      turn (angolo);
    END;
END;
BEGIN
  initturtle;
  assi (140, 95);
  poligono (140, 95, 6, 40);
  WHILE r < > 0 DO
    BEGIN
      read (n, r);
      initturtle;
      poligono (140, 95, n, r);
    END;
  textmode;
END.
```

3.3.5 - Tracciamento di un cerchio

Con il programma precedente, e ponendo $n = 100$, si ottiene una figura molto vicina ad un cerchio.

3.3.6 - Tracciamento di quadrati inseriti

Si vuole disegnare una serie di quadrati inseriti l'uno nell'altro come in figura:

Tracciato il primo quadrato, va calcolato l'angolo di rotazione rispetto ad esso e la lunghezza del lato.

Se il parametro *k*, che definisce l'angolo di rotazione, è tale per cui

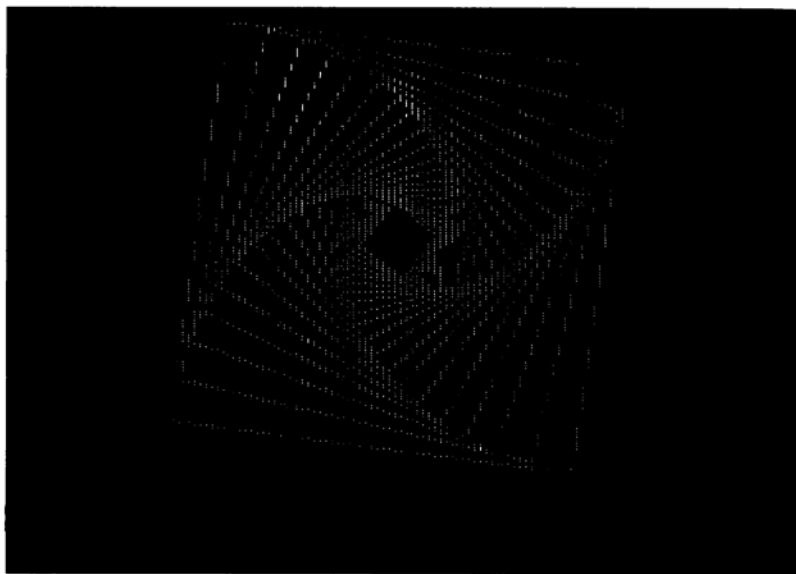
$$\text{tg } \alpha = \frac{BB'}{A'B} = \frac{1}{k-1}$$

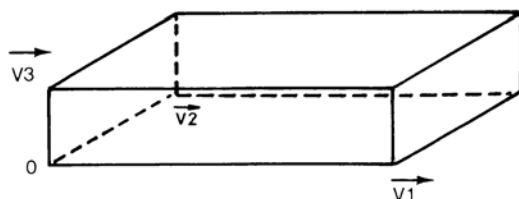
$$\alpha = \text{arcotg } \left(\frac{1}{k-1} \right)$$

e il nuovo lato è uguale a

$$\frac{\text{lato}}{k \text{ sen } \alpha}$$

allora si ha il programma seguente:

```
PROGRAM quadratinseriti;
uses turtlegraphics, transcend;
CONST
   l = 279;
   a = 191;
   pi = 3.14159;
VAR
   rot: real;
   k, lato: integer;
   i, j, n: integer;
BEGIN
   read (n, k, lato);
   initturtle;
   pencolor (none); moveto (60, 10);
   pencolor (violet);
   FOR j: = 1 TO n DO
   BEGIN
      FOR i: = 1 TO 4 DO
         BEGIN
            move (lato);
            turn (90);
         END;
      move (round(lato/k) );
      rot: = atan(1/(k -1) );
      turn (round(rot * 180/pi) );
      lato: = round(lato/(k * sin(rot) ) );
   END;
   textmode;
END.
```

3.3.7 - Figure nello spazio

Si debba disegnare un parallelepipedo

I vertici sono dati da combinazioni dei tre vettori $\vec{V1}$, $\vec{V2}$ e $\vec{V3}$, cioè ciascun vertice è tale per cui

$$\vec{OM} = b_1\vec{V}_1 + b_2\vec{V}_2 + b_3\vec{V}_3$$

con $b_i = 0$ o 1.

Quindi i vertici si possono rappresentare con terne di valori b_i.

Si può verificare che, partendo dal vertice (0,0,0), per disegnare un altro vertice si possono prendere tre direzioni. Invece, partendo da (1,0,0), da (0,0,1) o da (0,1,0), le direzioni possibili sono due.

Come regola generale, per un vertice (b_1, b_2, b_3) esistono tanti spigoli quante sono le b_i uguali a 0; la direzione dei vettori che partono dal vertice in questione corrisponde al vettore $\vec{V}_i$.

Così, ad esempio,

– dal vertice (0,0,1) partono spigoli equipollenti a $\vec{V}_1$ e $\vec{V}_2$;
– dal vertice (1,1,0) parte uno spigolo equipollente a $\vec{V}_3$.

Ricordiamo che un vettore $\vec{V}$ è caratterizzato dalle sue componenti V_x, V_y e V_z sugli assi delle x, delle y e delle z.

$$\vec{V} = V_x \cdot \vec{i} + V_y \cdot \vec{j} + V_z \cdot \vec{k}$$

ove $\vec{i}$, $\vec{j}$ e $\vec{k}$ sono i vettori unitari.

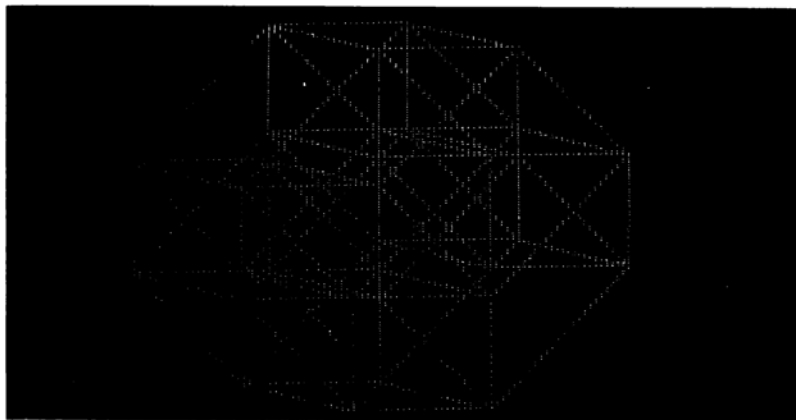
La somma di due vettori si ottiene sommando fra loro le corrispondenti componenti.

Questo metodo si può estendere al disegno di un iperparallelepipedo, partendo da n vettori: il numero dei vertici è allora 2^n.

Si otterrà così il programma che segue, nella prima parte del quale compaiono semplicemente le dichiarazioni e le procedure di conversione binaria.

```
PROGRAM parallelepipedo;
uses turtlegraphics;
TYPE
   binario = 0,1;
   bit = ARRAY[0 ... 15] OF binario;
   angolo = 0 ... 359;
   lung = 0... 99;
VAR
   x, y, x0, y0, i, k, n: integer;
   b: bit;
   a: ARRAY[0 ... 10] OF angolo;
   l: ARRAY[0 ... 10] OF lung;
FUNCTION potenza(b, j: integer): integer;
VAR
   p, i: integer;
BEGIN
   IF j = 0 THEN p: =1
      ELSE
      BEGIN
         p: = b;
         FOR i: = 2 TO j DO p: = p * b;
      END;
   potenza: = p;
END;
PROCEDURE bin (VAR b: bit; n, k: integer);
VAR
   i: integer;
```

```
BEGIN
  FOR i: = k - 1 DOWNTO 0 DO
    BEGIN
      IF n < potenza(2, i) THEN b[i]: = 0
        ELSE b[i]: = 1;
      n: = n - b[i] * potenza(2, i);
    END;
END;
PROCEDURE vertice (VAR x, y: integer);
VAR
  j: integer;
BEGIN
  pencolor (none); moveto (x0, y0);
  turn (360 - turtleang);
  FOR j: = 0 TO n - 1 DO
    BEGIN
      IF b[j] < > 0 THEN
        BEGIN
          turn (a[j]); move (l[j]);
          turn (360 - turtleang);
        END;
    END;
  x: = turtlex; y: = turtley;
END;
PROCEDURE disegno (x, y: integer);
VAR
  j: integer;
BEGIN
  pencolor (violet);
  turn (360 - turtleang);
  FOR j: = 0 TO n - 1 DO
    BEGIN
      IF b[j] = 0 THEN
        BEGIN
          turn (a[j]); move (l[j]);
          moveto (x, y);
          turn (360 - turtleang);
        END;
    END;
END;
BEGIN
  readln (n);
  FOR i: = 0 TO n - 1 DO read (a[i], l[i]);
```

```
  x0: = 100; y0: = 10;
  initturtle;
  FOR i: = 0 TO potenza(2, n) - 1 DO
    BEGIN
      bin (b, i, n);
      vertice (x, y);
      disegno (x, y)
    END;
END.
```

Nel programma compaiono due procedure, una di posizionamento ai vertici ed una di tracciamento degli spigoli che partono dai vertici.

3.3.8 - Sviluppi limitati di funzioni

Sappiamo che alcune funzioni di variabili reali possono essere espresse con approssimazione da serie polinomiali. Ad esempio,

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} \ldots + \frac{x^n}{n!} + \ldots$$

$$\frac{1}{1 + x} = 1 - x + x^2 - x^3 - x^3 - x^4 + \ldots \qquad \text{se } 0 < x < 1$$

$$\text{sen } x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots + (-1)^{n-1} \frac{x^{2+1}}{(2_n + 1)!}$$

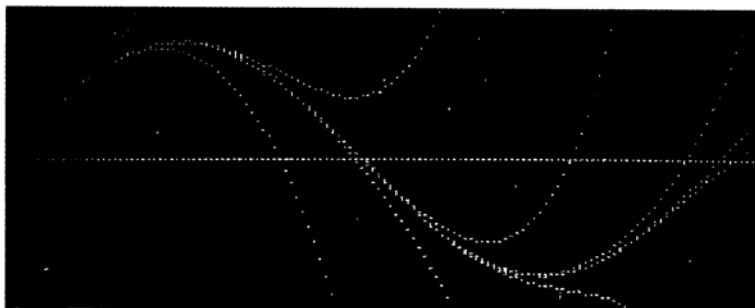
$$\log (1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} + \ldots \qquad \text{per } x < 1$$

È interessante verificare graficamente che questi polinomi danno luogo a curve sempre più approssimate alla funzione della quale sono approssimazione.

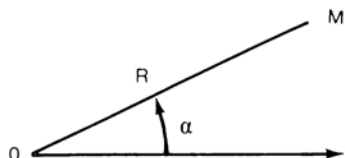
Ad esempio, per *sen x*, abbiamo il programma seguente:

```
PROGRAM approssimaz;
uses turtlegraphics;
VAR
  dx, dy, x: real;
  y, z, x0, y0: integer;
  n, i, k, s: integer;
FUNCTION potenza(b: real; j: integer): real;
VAR
  p: real;
  i: integer;
BEGIN
  IF j = 0 THEN p: =1
    ELSE
    BEGIN
      p: = b;
      FOR i: = 2 TO j DO p: = p* b;
    END;
  potenza: = p;
END;
FUNCTION fattor(n: integer): real;
VAR
  i: integer;
  f: real;
BEGIN
  f: = 1;
  FOR i = 1 TO n DO f: = f * i;
  fattor: = f;
END;
PROCEDURE punto (a, b: integer);
BEGIN
  pencolor (none);
  moveto (a, b);
  pencolor (violet);
  moveto (a, b);
END;
BEGIN
  readln (n);
  x0: = 0; y0: = 95; k: = 40;
  initturtle;
```

```
moveto (x0, y0);
  x: = 0; dx: = 0.025;
  WHILE x < 7 DO
  BEGIN
    y: = 0; s: = 1;
    FOR i: = 0 TO n DO
    BEGIN
      dy: = s * potenza(x, 2*i + 1)/fattor(2*i + 1);
      y: = y + round(k*dy);
      z: = round(k*x);
      punto (z, y + y0);
      s: = -s;
    END;
    x: = x + dx;
  END;
END.
```

Nel programma la formula per il calcolo di *sen x* tramite lo sviluppo limitato della relativa serie polinomiale è applicata direttamente per mezzo di una funzione potenza ed una funzione fattoriale. I fattori di scala dipendono dal sistema utilizzato.

3.3.9 - Disegni a coordinate polari

Alcune curve si esprimono molto più semplicemente per mezzo di coordinate polari, ove la distanza $r = OM$ è espressa in rapporto ad un'origine, ed in funzione dell'angolo α che r forma con l'asse originario.

R prende il nome di *raggio vettore*, α di *angolo polare*.

Una curva in coordinate polari è espressa quindi da una funzione del tipo

$$r = f(\alpha)$$

Esempi

1) Un cerchio con centro posto nel punto 0 è espresso, in coordinate polari, da *r = costante*.

2) $r = k(1 - \cos \alpha)$ è una curva che prende il nome di cuspide.

3) $r = k\alpha$ è una curva detta spirale di Archimede.

4) $r = k \operatorname{sen} 2\alpha$ è una rosa a quattro petali.

Il passaggio dalle coordinate polari alle coordinate cartesiane è molto semplice, perchè le coordinate x ed y del punto m sono tali per cui

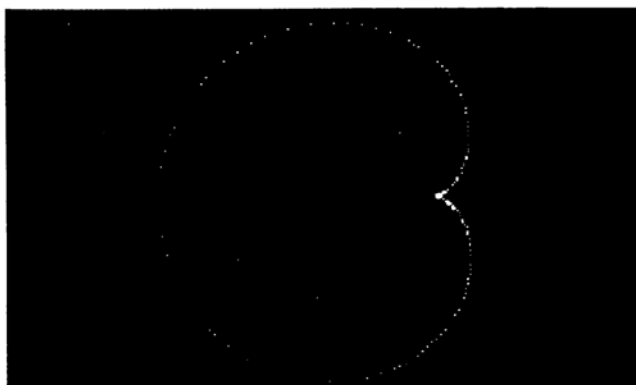
$$x = r \cos \alpha$$
$$y = r \operatorname{sen} \alpha$$

a) Supponiamo di voler tracciare la curva $r = k(1 - \cos \alpha)$; il programma sarà il seguente:

```
PROGRAM polare;
uses turtlegraphics, transcend;
CONST
  pi = 3.14159;
TYPE
  angolo = 0 ... 359;
VAR
  teta, ang: angolo;
  raggio: real;
  k, x0, y0: integer;
BEGIN
  x0: = 140; y0: = 95;
  read (k);
  initturtle;
  ang: = 0;
  teta: = 5;
  WHILE ang < 360 DO
    BEGIN
      pencolor (none);
      moveto (x0, y0);
```

```
        raggio: = k * (1 - cos(ang * pi/180) );
        move (round(raggio) );
        pencolor (green);
        moveto (turtlex, turtley);
        ang: = ang + teta;
        turnto (ang);
     END;
END.
```

Con la grafica della tartaruga non si deve passare necessariamente alle coordinate cartesiane, ma, per ottenere il raggio vettore, si deve sempre tornare in *x0*, *y0*, dopo ogni movimento.

Avremo una curva quale quella che appare nella figura seguente:

b) Possiamo dimostrare che la famiglia di curve la cui equazione polare è

$$r = k + \cos n\alpha$$

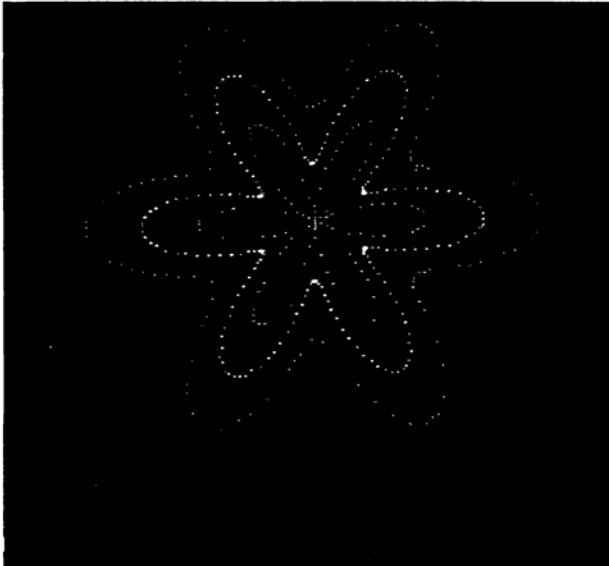
rappresenta la proiezione della traiettoria di un punto che si sposta regolarmente su un toro.

Queste curve hanno la forma di rosoni, ovvero di fiori.

Il programma è il seguente:

```
PROGRAM polare;
uses turtlegraphics, transcend;
CONST
   pi = 3.14159;
TYPE
   angolo = 0... 360;
VAR
   teta, ang: angolo;
   raggio: real;
   x, y, n, k, x0, y0: integer;
```

```
BEGIN
  x0: = 140; y0: = 95;
  read (n);
  initturtle;
  FOR k: = 1 TO 3 DO
  BEGIN
    ang: = 0;
    teta: = 1;
    WHILE ang < 360 DO
      BEGIN
        pencolor (none);
        moveto (x0, y0);
        raggio: = 20 *(k + cos(n * ang * pi/180) );
        move (round(raggio) );
        pencolor (green);
        moveto (turtlex, turtley);
        ang: = ang + teta;
        turnto (ang);
      END;
    pencolor (none);
    moveto (k, k); wchar ('k');
  END;
END.
```

Con questo programma si ottengono curve simili a quelle della figura seguente:

4 - CONCLUSIONI

Spesso la grafica è considerata un lusso, o un "giocattolo" inutile per chi impara a programmare.

Nel primo caso, vengono poste sotto accusa le periferiche molto costose, nel secondo i procedimenti semigrafici utilizzati per i giochi video.

Ma lo sviluppo dei sistemi grafici funzionanti su un normale televisore ha fatto sì che la grafica non fosse più un lusso, e potesse essere anzi utilizzata non solo per i giochi, ma anche per programmi seri.

Essa permette infatti di presentare, ad esempio, i risultati in una maniera molto più espressiva che con delle tabelle di cifre.

Da ultimo, la grafica ha una qualità fondamentale che nessun altro dispositivo di uscita è in grado di offrire: la capacità di rappresentare il tempo in forma di movimento. Questa importantissima proprietà sarà estremamente utile non soltanto nel campo dei giochi, ma anche nella rappresentazione dinamica dei risultati: simulazione di processi dinamici, animazione e rappresentazione di forme artistiche o geometriche.

ESERCIZI

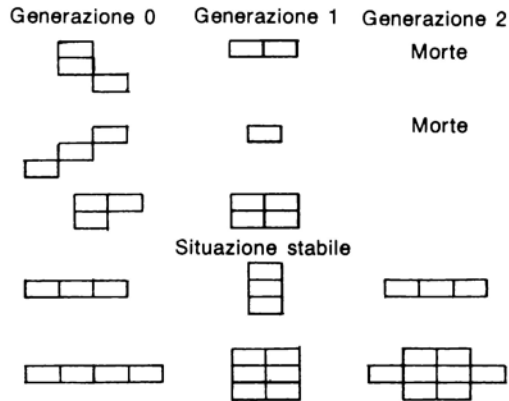
1. Scrivere un programma che legga un punto, e poi lo cancelli spostandolo.

2. Il gioco della vita.
   Il matematico Conway ha ideato un gioco che ha chiamato gioco della vita, basato su delle "cellule" capaci di riprodursi, scomparire o sopravvivere, obbedendo a determinate leggi, dette "genetiche".
   Le cellule sono rappresentate da elementi posti su una scacchiera di dimensioni arbitrarie. Quindi ogni cellula è circondata da otto caselle che possono contenere altre cellule.
   Valgono le seguenti leggi:

   a) *Sopravvivenza*
      Tutte le cellule che hanno due o tre cellule adiacenti sopravvivono fino alla generazione successiva.

   b) *Morte*
      Tutte le cellule che hanno quattro (o più) cellule adiacenti scompaiono, ovvero muoiono, per sovrappopolamento. Le cellule con una sola, o nessuna cellula adiacente muoiono per isolamento.

   c) *Nascita*
      L'esistenza di tre cellule adiacenti, comunque disposte, fa nascere una nuova cellula per la generazione successiva.

È importante notare che tutte le nascite e tutte le morti relative ad una generazione avvengono contemporaneamente.

Scrivere il relativo programma.

3. Il gioco del quadrato cinese.

1	2	3
4	5	6
7	8	9

   Il giocatore dispone delle pedine (ad esempio asterischi e trattini) nel tentativo di formare una linea di tre punti, mentre l'avversario cerca d'impedirglielo. Le caselle del quadrato sono numerate come appare in figura.
   Scrivere il relativo programma.

4. Le torri di Hanoi.
   All'inizio del gioco, si ha una torre formata da dischi di diametro decrescente.
   Questi dischi vanno spostati in modo da formare un'altra torre, nella quale i diametri dei dischi siano via via più piccoli man mano che la torre cresce.
   Non possono esserci contemporaneamente più di tre pile, o torri, di dischi.
   Visualizzare le torri utilizzando il programma presentato nel Capitolo 5.

CAPITOLO 9

LE ESTENSIONI DEL PASCAL SUI MICROCALCOLATORI

Abbiamo già visto un certo numero di estensioni del sistema Pascal U.C.S.D.

Altri sistemi Pascal, realizzati su microcalcolatori, dispongono di estensioni non standard. Qui ne presenteremo alcune, tipiche del sistema Apple, che dispone del Pascal U.C.S.D.: per accedere ad esse occorre introdurre la clausola *uses Applestuff*, propria dell'Apple.

1 - DISPOSITIVI PARTICOLARI D'INGRESSO/USCITA

In aggiunta ai dispositivi d'ingresso/uscita standard, nel sistema Apple esiste la possibilità di collegarsi a dei dispositivi semplici, come manopole (paddles), pulsanti, altoparlanti e dispositivi TTL compatibili (TTL: dall'inglese Transistor Transistor Logic).

1.1 - Le manopole (paddles)

Si tratta di pulsanti a moto circolare continuo, collegati a potenziometri che trasmettono un valore analogico anch'esso continuo: questo valore può essere associato ad un numero intero, su un byte. Abbiamo così un dispositivo interattivo più adatto della tastiera per alcuni giochi.

Questi dispositivi si chiamano in inglese paddles, cioè manopole. Ad essi si accede richiedendo una funzione il cui parametro specifica il dispositivo selezionato, fino ad un massimo di quattro.

La sequenza di chiamata, molto semplice, è pertanto

```
x: = paddle(i)
```

ove *i* è un intero compreso fra 0 e 3, o anche un intero modulo 4.

La combinazione di due di questi dispositivi permette, in particolare, di associare due coordinate, cioè un punto, a ciascuna manopola; ad esempio,

```
x: = paddle(0);
y: = paddle(1);
```

La funzione *paddle* assume valori interi compresi fra 0 e 255.
Generalmente, fra due chiamate bisogna effettuare dei cicli di attesa del tipo

```
FOR i: = 0 TO n DO;
```

1.2 - I pulsanti

Associati alle manopole, i pulsanti permettono di convalidare un'operazione binaria: a tal fine si ricorre alle funzioni booleane *button(i)*, con $i = 0, 1, 2$ per i pulsanti veri e propri. Se il pulsante è premuto, il valore corrispondente è vero, altrimenti sarà falso.
La funzione *button(3)* è riservata alla lettura delle cassette magnetiche.

1.3 - Le uscite TTL

Queste uscite permettono di controllare apparecchi esterni a logica TTL; in questo modo si possono comandare da programma apparecchiature elettroniche semplici, quali ad esempio i trenini elettrici.
Queste operazioni vengono attivate da una procedura denominata *ttlout* (cioè uscita TTL), che contiene due parametri: il primo è il numero dell'uscita selezionata (da 0 a 3), il secondo è un valore logico (vero o falso):

```
ttlout (i, logico)
```

Se il valore *logico* è vero, l'uscita è attiva, fino al momento in cui un'altra chiamata della procedura invierà un valore logico falso.

1.4 - Esame della tastiera

Con i microcalcolatori accade molto di frequente che un programma interattivo esegua dei cicli all'infinito, finché non viene premuto un tasto: la verifica fatta sul carattere *EOF*, ottenuto battendo sulla tastiera *CONTROL C*, non sempre risponde allo scopo. Esiste pertanto una funzione particolare che permette di sapere se un tasto è stato premuto, e/o se è presente nella memoria tampone associata alla tastiera: questa funzione è molto pratica perché, come si è accennato, permette di far riciclare un programma finché non viene battuto un carattere.
Questa funzione, denominata *keypress*, è una funzione booleana: è vera se è stato battuto un tasto.
Con essa si possono usare molto bene le strutture while e repeat.

Esempi

1) WHILE NOT keypress DO
   Ciclo del programma;

2) REPEAT
   Ciclo del programma
   UNTIL keypress

1.5 - L'uscita altoparlante

Una procedura di attivazione di un piccolo altoparlante può essere utilizzata nei giochi e nei programmi interattivi, allo scopo di attirare l'attenzione dell'utilizzatore. Si tratta della procedura

nota (suono, durata)

Il primo parametro è un intero che rappresenta l'altezza del suono in frequenza; può variare da 0 a 50, con due valori, 0 e 1, particolari:
– 0 indica una pausa;
– *nota (1, 1)* produce un suono secco (clic).

I valori compresi fra 2 e 48 corrispondono approssimativamente ad una scala cromatica.

Il secondo parametro indica la durata del suono: *durata* è un parametro intero che va da 0 a 255.

Esempio

Creazione di una scala cromatica:

```
PROGRAM scala;
uses Applestuff;
CONST
  grave = 0;
  acuto = 50;
VAR
  suono: grave ... acuto;
  durata: 0 ... 255;
BEGIN
  durata: = 200;
  FOR suono: = grave TO acuto DO
    nota (suono, durata);
END.
```

La tastiera permette anche di simulare un "pianoforte". Prendiamo ad esempio le lettere dell'alfabeto per rappresentare le note (a = do, b = re, c = mi, e così via). Supporremo che la durata delle note sia costante. Ed ecco il programma, nel quale il do corrisponde al valore 2 della procedura *nota*.

```
PROGRAM pianoforte;
uses Applestuff;
CONST
  grave = 0;
  acuto = 50;
VAR
  suono: grave ... acuto;
  durata: 0 ... 255;
  tasto: char;
BEGIN
  durata: = 200;
  WHILE NOT eof DO
    BEGIN
      read (tasto);
      suono: = 2 + ord(tasto) – ord('a');
      nota (suono, durata);
    END;
END.
```

Introducendo ad esempio i caratteri aaacecaecca, avremo la melodia di *Al chiar di luna*.

Inutile dire che si tratta di uno strumento rudimentale.

1.6 - Procedure d'ingresso/uscita specifiche

Da ultimo, alcune procedure permettono di lavorare ad un livello prossimo ai dispositivi d'ingresso/uscita.

Le segnaliamo semplicemente a titolo informativo, perché, se non si prendono le debite precauzioni, o non si ha sufficiente esperienza, usarle può essere pericoloso. Intendiamo parlare di tutte le procedure del tipo *unitxxxx*, che sono:

unitbusy: verifica se una periferica è occupata;
unitwait: aspetta che l'unità sia libera;
unitclear: azzera la periferica;
unitread: lettura;
unitwrite: scrittura.

1.7 - I flussi non standard

Il sistema Pascal U.C.S.D. dispone di un tipo di flusso non strutturato che permette di leggere, o scrivere, un flusso a blocchi, senza ricorrere a variabili finestra o componente.

Questi flussi vengono dichiarati tali, ma non si utilizzano le procedure *get* e *put*, bensì le procedure *blockread* e *blockwrite,* che permettono di leggere e scrivere direttamente dei blocchi di dati in una zona di memoria tampone (buffer) di 512 bytes.

Queste funzioni sono di tipo intero, e si limitano a far ritornare il numero dei blocchi trasferiti, per quanto abbiano diversi parametri d'ingresso:

blockread (nomeflusso, vettore, nblocco, noblocco)

Il primo parametro è il nome del flusso; il secondo è un vettore la cui dimensione dev'essere un multiplo di 512; gli altri sono rispettivamente il numero dei blocchi ed il numero d'ordine del blocco relativamente all'inizio del flusso: lettura e scrittura vanno effettuate con riferimento a quest'ultimo parametro. Se non viene specificato il parametro *noblocco*, le operazioni d'ingresso/uscita avranno luogo secondo l'ordine sequenziale.

La funzione *eof* diventa vera alla lettura dell'ultimo blocco del flusso.

Queste procedure sono consigliabili quando si vuol accedere più rapidamente al flusso.

Presentiamo ora due programmi che permettono rispettivamente di leggere e di scrivere delle griglie di selezione da disco: saranno molto utili se, ad esempio, si vuole che l'accesso alle griglie sia più rapido di quello standard. Comunque è bene sottolineare che val la pena di ricorrere a queste procedure solo quando si debbano migliorare le prestazioni di un sistema, oppure quando la rapidità di accesso al flusso è un fattore decisivo. All'inizio è comunque preferibile utilizzare procedure standard, allo scopo di ottenere una versione trasportabile.

1) Scrittura di una griglia

```
PROGRAM griglia;
CONST l1 = 3; l2 = 23;
VAR i, j: integer;
  selezione: ARRAY[0 ... 23] OF string;
  nomegrigl: string;
  nogriglia, btras: integer;
  car: char;
  g: file;
BEGIN
  write ('no di griglia?');
  read (nogriglia);
  write ('volume:'); readln (selezione[0]);
  rewrite (g,'griglia.data');
  while nogriglia > = 0 DO
```

```
    BEGIN
      page (output);
      writeln ('      ', selezione[0]);
      FOR i: = 1 TO l1 DO writeln;
        j: = l1;
          REPEAT
            j: = j + 1;
            i: = j - l1;
            write (i, ' ');
            readln (selezione [i]);
          UNTIL (selezione [i] = ' ') OR (j = l2);
      btras: = blockwrite (g, selezione, 1, nogriglia);
      page (output);
      write ('no griglia?'); read (nogriglia);
    END;
  END.
```

2) Lettura di una griglia

```
PROGRAM leggeregriglia;
CONST l1 = 3; l2 = 23;
VAR i, j: integer;
  selezione: ARRAY[0 ... 23] OF string;
  nogriglia, brats: integer;
  car: char;
  g: file;
BEGIN
  write ('no di griglia?');
  read (nogriglia);
  WHILE nogriglia > = 0 DO
  BEGIN
    page (output);
    reset (g, 'griglia.data');
    btras: = blockread (g, selezione, 1, nogriglia);
    close (g, lock);
    writeln ('      ', selezione[0]);
    FOR i: = 1 TO l1 DO writeln;
      j: = 1;
      WHILE selezione [j] < > ' ' DO
        BEGIN
          writeln (j, ' ', selezione[j]);
          j:= j + 1
        END;
      write ('ok'); readln (car);
```

```
      page (output);
      write ('no griglia?'); read (nogriglia);
    END;
  END.
```

2 - LE ESTENSIONI MATEMATICHE

2.1 - La funzione generatrice di un numero casuale

Questa funzione permette di ottenere un valore compreso fra 0 e 32767; si chiama semplicemente *random* (cioè casuale), e non ha alcun parametro.

I numeri che si ottengono sono sempre gli stessi, dal momento che vengono generati da una sequenza ripetitiva.

Con la procedura *randomize* invece si ottiene ogni volta una sequenza diversa.

Ecco qualche esempio di utilizzo:

1)
```
PROGRAM casuale;
uses transcend, Applestuff;
VAR
  n, i: integer;
BEGIN
  readln (n);
  randomize;
  FOR i: = 1 TO n DO
    writeln (random);
END.
```

Con la funzione *randomize* si ha una sequenza diversa ad ogni esecuzione.

Esecuzione (Numeri casuali)

Questa sequenza è ottenuta non ricorrendo a *randomize*.

```
10 Ⓡ
9305
22934
3922
10796
32327
13781
9407
20890
32338
6184
```

2) Si vuole generare un numero casuale compreso fra due valori a e b ($b > a$).
L'intervallo di variazione è $b - a$; allora, essendo a il valore iniziale, basta definire

n = a + random mod(b − a + 1)

Un'applicazione è data dall'estrazione di numeri a caso, avendo specificato quali devono essere i due estremi:

```
PROGRAM numerocasuale;
uses transcend, Applestuff;
VAR
  n, i: integer;
  a, b: integer;
FUNCTION casu(min, max: integer): integer;
BEGIN
  casu: = min + random MOD (max − min + 1)
END;
BEGIN
  writeln ('numero?'); readln (n);
  writeln ('limiti:');
  read (a, b);
  randomize;
  FOR i: = 1 TO n DO
    write (casu(a, b),' ')
END.
```

Il programma può essere usato, in particolare, per simulare una partita ai dadi ($a = 1$, $b = 6$), una giocata al lotto ($a = 1$, $b = 90$), etc.

2.2 - Gl'interi lunghi

Con il sistema Apple Pascal si possono usare numeri interi di lunghezza maggiore di quella standard: si ricordi che per gl'interi standard il valore massimo è 32767 in valore assoluto, per 16 bits.

Come? Definendo dei tipi *interi*, di cui va specificata la lunghezza in cifre decimali.

Il numero massimo di cifre è 36, per cui si ha la possibilità di rappresentare dei numeri molto grandi, generalmente trattati come reali.

La dichiarazione relativa indica il tipo intero, seguito dal numero di cifre desiderato, messo fra parentesi quadre.

Esempi

```
1)      TYPE
           interolungo: integer[10];
        VAR
           x: interolungo;
```

oppure

```
2)      VAR
           r = integer[15];
```

Le istruzioni di assegnazione fra interi lunghi sono possibili; si possono anche assegnare degl'interi semplici a degli interi lunghi, ma, chiaramente, non vale il contrario.

Ed ecco un esempio di calcolo di numeri casuali su dieci cifre: un metodo di generazione di numeri pseudocasuali consiste nell'usare una formula di congruenza del tipo

$$r_n = ar_{n-1} \text{ modulo } n$$

ove *a* ed *n*, nonchè r_0, sono scelti in modo da ottenere una sequenza che possieda le proprietà statistiche volute (numero pseudocasuale).

In particolare, per generare dei numeri di dieci cifre decimali, useremo:

$$r_n = 100003\ r_{n-1} \text{ (modulo } 10^{10})$$

r_0 può essere un qualsivoglia numero dispari non divisibile per 5.

Per ottenere dei numeri compresi fra 0 ed 1, definiremo:

$$u_n = r_n \cdot 10^{-10}$$

Con r_0 = 123.456.789, si ha:

u1 = 0.6049270367...

Questo è il programma corrispondente:

```
PROGRAM numerocasuale;
uses transcend, Applestuff;
TYPE
   interolungo = integer[20];
VAR
   n, i: integer;
   r: interolungo;
PROCEDURE casuale (VAR gn: interolungo);
VAR
   ca, m, g0, a: interolungo;
```

```
BEGIN
   m: = 10000000000;
   a: = 100003;
   g0: = 123456789;
   IF gn = 0 THEN gn: = g0;
   gn: = a * gn;
   ca: = gn div m;
   gn: = gn - ca * m;
END;
BEGIN
   r: = 0;
   readln (n);
   FOR i: = 1 TO n DO
      BEGIN
         casuale (r);
         writeln (r);
      END;
END.
```

In questo programma ci siamo avvalsi di una peculiarità che non è presente su tutti i sistemi Pascal: la possibilità di definire degl'interi lunghi (fino a 36 cifre). Questo semplifica il programma: la sequenza di numeri che segue è quanto si ottiene all'esecuzione.

Esecuzione

1	60492	70367
2	51845	11101
3	66636	33303
4	33211	99909
5	99544	99727
6	98361	99181
7	94266	97543
8	80343	92629
9	33660	77887
10	78869	33661
11	70269	00983
12	11790	02949
13	38319	08847
14	23804	26541
15	97953	79623
16	73483	38869
17	59322	16107
18	94573	49821
19	33541	49463
20	50087	48389

1. Modificare il programma del paragrafo 1.5 al fine d'introdurre la durata delle note: 4 corrisponderà ad una semibreve, 2 ad una minima, 1 ad una semiminima, ½ ad una croma. Con un uguale procedimento si potrà avere l'ottava.

2. Scrivere un programma che introduca una partitura, e poi suoni la melodia relativa.

3. Scrivere un programma che giochi a testa e croce. Stampare i risultati sotto forma di una sequenza di *t* e di *c*, conteggiando il numero di "testa" e di "croce" per un dato numero di lanci.

4. Scrivere un programma che stampi una lista di numeri a caso, badando a che uno stesso numero non esca due volte.

5. Applicare l'esercizio precedente al gioco del lotto.

CAPITOLO 10

CREAZIONE E MESSA A PUNTO DEI PROGRAMMI IN PASCAL

1 - CREAZIONE DEI PROGRAMMI PASCAL

Abbiamo visto a più riprese che il Pascal è un linguaggio strutturato; di conseguenza permette di creare programmi modulari strutturati a blocchi di funzioni e di procedure.

Per far questo si ricorre ad una tecnica di approssimazioni successive, definendo a livello del programma principale i moduli necessari, cioè le procedure o le funzioni da impiegare. Ciascun modulo può a sua volta venir scomposto in procedure o in funzioni, che devono essere definite in maniera indipendente e verificate isolatamente, cercando di evitare l'uso di variabili globali, che rischierebbero di produrre spiacevoli effetti margine.

Ogni blocco poi dovrà essere strutturato evitando l'impiego di istruzioni di diramazione.

I programmatori abituati ai linguaggi non strutturati, come il BASIC e il FORTRAN, possono avere delle difficoltà iniziali; ma è un problema destinato a scomparire presto, non appena si prende familiarità con le strutture IF..THEN..ELSE, WHILE e REPEAT. In questo libro non abbiamo mai usato istruzioni di diramazione, e questo prova che il loro impiego è assolutamente eccezionale.

Per quanto riguarda l'algoritmica, il linguaggio Pascal permette di definire procedure e funzioni ricorsive. Val la pena di ricorrere a questo metodo di definizione ogni volta che il problema vi si presta: la programmazione richiederà un lavoro molto minore, ed i programmi saranno notevolmente più semplici e chiari.

Bisogna poi servirsi largamente del concetto di *tipo*, che è un aspetto fondamentale del linguaggio.

Una raccomandazione per il programmatore alle prime armi sulla quale val la pena d'insistere è di non avventurarsi nella programmazione prima di un'analisi accurata del problema. Agl'inizi sarà opportuno scrivere delle procedure molto semplici a titolo

di verifica di questa o quella parte dell'algoritmo; e non esitate a ricominciare tutto da capo se il programma s'ingarbuglia e diventa mal strutturato.

Come ultima osservazione, per i programmatori di professione è importante definire delle procedure parametriche, da poter poi utilizzare in altri programmi.

Queste poche indicazioni sono semplicemente una guida di carattere generale: solo con la pratica si diventa programmatori esperti, qualunque sia il linguaggio.

Resta comunque il fatto che il Pascal permette agli utilizzatori più smaliziati di creare un programma in modo più soddisfacente, non solo, ma anche di dimostrarne l'esattezza ancor prima di realizzarlo.

Ma questo ci porta fuori dei limiti di quest'opera, destinata esclusivamente ai principianti; anche così comunque è necessario soffermarsi sulla messa a punto dei programmi.

2 - LA MESSA A PUNTO DEI PROGRAMMI

In Pascal, la messa a punto dei programmi, se questi sono ben strutturati, è relativamente facile, tanto che in moltissimi casi non richiede il ricorso ai programmi specificatamente destinati alla messa a punto, i cosiddetti programmi di *debugging.*

Infatti gli errori di sintassi s'individuano e si correggono facilmente. Quanto agli errori di esecuzione, messaggi di errore ne segnaleranno la natura: in appendice è riportata la lista degli errori standard.

Quando non si riesce ad individuare la causa di un errore, può essere necessario inserire delle istruzioni che portino alla stampa di risultati intermedi subito prima dell'istruzione che dà luogo all'errore: queste istruzioni saranno poi eliminate una volta corretto l'errore.

Su alcuni sistemi si dispone comunque di programmi per la messa a punto (programmi di debugging), che forniscono la stampa e la visualizzazione delle istruzioni man mano che procede l'esecuzione del programma.

In alcuni casi si possono definire all'interno del programma dei punti di arresto (*break-points* in inglese), che permettono di verificare la parte di programma già eseguita prima di riprendere l'esecuzione, eventualmente passo passo, cioè istruzione per istruzione.

Queste facilitazioni sono disponibili solo sui sistemi di tipo professionale. Servono soprattutto con le versioni di Pascal non interattive: infatti, quando un programma è stato giudicato corretto dal programma compilatore, può essere necessario ricorrere a questi strumenti per la messa a punto. Sono invece molto meno utili con le versioni di Pascal interattive, perché in questo caso è sempre possibile inserire delle istruzioni di uscita.

Prima di andare avanti, riteniamo opportuno sottolineare che il programmatore non deve prendere l'abitudine di mettersi a scrivere un programma prima di averlo analizzato accuratamente sulla carta. Qui sta uno dei rischi dei sistemi interattivi: una cattiva analisi porta immancabilmente a cattivi programmi.

Se una prima versione non funziona, è senz'altro possibile correggerla direttamente, ma, se sono necessarie troppe correzioni, è molto meglio riprendere il programma, o la parte di programma incriminata, e riscriverlo sulla carta per avere una versione definitiva, chiara e ben strutturata. Questa precauzione apparirà in tutto il suo peso quando un programma è destinato ad essere utilizzato, o eventualmente modificato, in seguito.

2.1 - I principali tipi di errore e la loro interpretazione

Un grosso problema è che molti programmatori restano disorientati, a volte fino al punto di cedere le armi, quando non riescono a capire la natura degli errori che commettono; né il loro compito è facilitato dalla laconicità dei messaggi d'errore, assolutamente inadatti a specificare quello che si è potuto verificare a livello di linguaggio evoluto, e non a livello di sistema. A questo si aggiunga il fatto che i messaggi molto spesso sono in inglese, dal che sorgono per alcuni problemi di comprensione o d'interpretazione.

Questo libro si propone non di fornire un'interpretazione dei messaggi d'errore dei vari sistemi, ma di prevenire l'utilizzatore sugli errori più frequenti, spiegando inoltre come cercarli. La lista degli errori standard è presentata nell'Appendice 4.

2.1.1 - Gli errori di sintassi

Sono errori relativamente facili da trovare. Prima di tutto bisogna esaminare uno per uno tutti gli elementi dell'istruzione. Sono stati dichiarati tutti gl'identificatori? Le parole-chiave sono scritte esattamente? I nomi delle variabili e delle costanti sono corretti? Ci sono, e al posto giusto, i separatori (spazio bianco, punto, virgola, apice, etc.)? I BEGIN e gli END sono in numero uguale?

Nel caso delle istruzioni complesse, bisogna studiarne la struttura: si sono per caso dimenticati, nelle istruzioni aritmetiche, dei parametri, degli operatori o dei separatori? Non si sono inserite indebitamente anche delle variabili non numeriche? Sono stati esattamente rispettati i tipi in entrambi i membri di un'assegnazione o di un'espressione? Nelle istruzioni di selezione IF, bisogna innanzitutto verificare l'istruzione condizionale: gli operatori relazionali o logici sono messi al posto giusto? Poi si esaminerà l'istruzione che fa seguito alla clausola THEN, poi ancora quella che segue la clausola ELSE. Nelle istruzioni FOR bisogna verificare le espressioni utilizzate per i valori iniziali e finali. Nelle strutture WHILE e REPEAT occorre verificare la condizione di uscita dai cicli. Nelle istruzioni CASE si verificheranno i valori del selettore e le elaborazioni relative a ciascun caso.

Negli altri casi, si verificherà che i separatori che compaiono nelle istruzioni d'ingresso-uscita siano collocati al posto giusto, e si accerterà pure che le variabili indicizzate siano state dichiarate e dimensionate in una dichiarazione di tipo.

Le strutture dinamiche con puntatori vanno esaminate accuratamente.

2.2 - Gli errori di esecuzione

Distinguiamo due casi: quello in cui il programma è già stato eseguito, e quello in cui non lo è ancora stato.

Se il programma non è mai stato eseguito, si potrà avere un errore di logica, a livello di analisi o di scrittura del programma. Se il programma non fornisce nessun risultato, sarà bene verificare se per caso esegue un ciclo infinito. Se invece fornisce risultati errati, si cercherà di localizzare il punto in cui si trova l'errore.

Nel caso di calcoli, si verificherà che le istruzioni aritmetiche siano corrette, ed in particolar modo che la gerarchia degli operatori sia rispettata.

Se il programma fa "il contrario" di quello che dovrebbe, si verificherà che nelle istruzioni di test le condizioni siano definite esattamente.

Occorre anche verificare se l'errore non è semplicemente a livello delle operazioni d'ingresso/uscita.

Quando intervengono dei cicli FOR, WHILE o REPEAT, bisogna verificare i limiti e le condizioni di uscita, e soprattutto badare ai problemi legati agl'intervalli: un giro in più o in meno all'interno di un ciclo può dare risultati aberranti.

Verificato che tutti questi punti non nascondono errori, si inseriranno delle istruzioni di uscita per determinare con esattezza il gruppo d'istruzioni a partire dal quale i risultati sono errati.

Comunque non bisogna abusare di questa tecnica: è dall'esame del programma che si devono poter individuare gli errori.

Nel caso invece di programmi che sono già stati eseguiti, gli errori che eventualmente intervengono dopo varie esecuzioni dipendono essenzialmente dal fatto che i dati in gioco cambiano da un'esecuzione all'altra. Si possono avere due casi: nel primo, ci sono dei dati particolari che danno il controllo a rami del programma non ancora percorsi, rivelando così errori di programmazione non individuati. A volte poi non è stata prevista la possibilità di questi dati particolari, e ciò provoca errori di esecuzione: superamento dei limiti (*overflow*), divisione per zero, etc. Pertanto è indispensabile predisporre delle serie di dati di test il più possibile estese, e considerare tutti i casi particolari.

Nel secondo caso si ha un funzionamento aberrante a causa di dati errati o di operazioni sbagliate da parte dell'utilizzatore. Gli errori di questo tipo sono di solito imprevedibili, ma in ogni caso permettono di mettere alla prova la completezza del programma. Infatti, qualunque sia il problema da risolvere, se il programma è destinato ad essere utilizzato da altri, oltre che da chi l'ha scritto, è fondamentale predisporre delle protezioni, e in particolar modo dei messaggi di errore tali da coprire tutte le possibili manovre errate e le scelte erronee dei dati. Può anche capitare che il corpo principale del programma sia formato da un numero d'istruzioni inferiore a quello destinato alla gestione di errori di questo tipo!

Si tratta, dunque, di un lavoro abbastanza ingrato, ma indispensabile se si vogliono avere programmi completi e capaci di resistere a qualsiasi prova.

Quindi, prima di mettere in servizio un programma, è opportuno metterlo nelle mani di una persona inesperta, per verificarne la completezza. E si tenga presente che

l'utente ha sempre ragione. Inutile cercare di difendersi adducendo il fatto che non è stato rispettato il manuale d'uso, o che è stata fatta una manovra sbagliata: un programma dev'essere protetto in tutti i modi contro ogni manipolazione errata.

Per finire, sottolineiamo che è pure necessario predisporre dei messaggi d'errore che siano sufficientemente espliciti per l'utilizzatore; questo vale soprattutto per i programmi interattivi.

3 - CONCLUSIONI

In questo libro abbiamo illustrato le caratteristiche e le peculiarità del linguaggio Pascal. I primi capitoli concernono lo studio delle strutture e delle istruzioni standard del linguaggio, e illustrano altresì i principi e le tecniche di base della programmazione in modo progressivo, in modo da permettere ai principianti di imparare più facilmente e rapidamente a programmare.

Il linguaggio Pascal è quanto mai adatto per chi voglia imparare un linguaggio algoritmico moderno, e prepara all'utilizzo dei linguaggi futuri, come il linguaggio ADA.

Inoltre, essendo disponibile su microcalcolatori indipendenti, permette a professionisti ed insegnanti di scrivere e mettere a punto rapidamente programmi per verificare un algoritmo, visualizzare una curva, consultare ed aggiornare piccoli archivi, e così via.

Negli ultimi capitoli abbiamo introdotto delle estensioni del linguaggio in molti casi non standardizzate, illustrando volutamente quelle disponibili su piccoli calcolatori. Anche qui quel che conta è avere ben capito i concetti di base: il concetto di primitive di un sistema di flussi, la generazione di vettori e di figure, il cambiamento di origine e di scala nel campo della grafica.

È chiaro che un sistema potrà realizzare questi concetti in modo più flessibile di un altro; ma gli esempi che si sono portati qui si proponevano semplicemente di far conoscere le possibilità offerte da un sistema di prestazioni minime.

Contrariamente a quanto si pensa di solito, i sistemi microcalcolatore che chiamiamo personali non sono quelli che offrono le più basse prestazioni, né i meno flessibili. Certo, mancano talora di protezioni, ed offrono molto spesso dei sistemi di gestione di flussi elementari, ma anche qui assistiamo ad una rapida evoluzione, determinata dalla possibilità di ammortizzare un software più sofisticato su un elevato numero di unità.

Ci auguriamo quindi che il libro possa soddisfare le attese di tutti i neoutilizzatori, e che permetta loro non solo di padroneggiare fino in fondo il linguaggio Pascal, ma anche di sviluppare applicazioni proprie e idearne altre più complesse.

BIBLIOGRAFIA

1. *Apple Pascal, Reference Manual*, Apple Computer, 1979.

2. Arsac, J., *La construction de programmes structurés*, Dunod, 1977.

3. Bowles, K. L., *Programming in Pascal,* Springer Verlag, 1977.

4. Jensen, K., Wirth, N., *Pascal. Manuale e standard del linguaggio,* Gruppo Editoriale Jackson, 1981.

5. Knuth, D., *The Art of Computer Programming*, Vol. 1, 2, 3, Addison Wesley.

6. Nassi, I., Schnneidermann, *Iteration graphs*, in ACM Sigplan, Vol. 8, n. 8, Aug. 1973, pag. 12-26.

7. Wirth, N., *Algorithm and Data Structures*, Prentice Hall, 1976.

APPENDICE 1

CENNI SULLA NUMERAZIONE BINARIA

Definizione di linguaggio binario

Il linguaggio binario è quel linguaggio il cui alfabeto è formato da due caratteri:

$$A = \{0, 1\}$$

Quindi le parole di questo linguaggio saranno 0, 1, 10, 11, 100, 101, 110, 111, e così via.

Il linguaggio binario può essere utilizzato come sistema di codifica di un insieme.

Esempio

$$0 \leftrightarrow a_1$$
$$1 \leftrightarrow a_2$$
$$10 \leftrightarrow a_3$$

Il sistema di numerazione binaria

Il linguaggio binario si può usare anche come *sistema di numerazione*, associando a ciascun carattere binario un valore uguale ad una potenza di 2: cioè alla posizione più a destra si associa la potenza 0, a quella che segue procedendo verso sinistra la potenza 1, poi la potenza 2, e così via. Questo è quello che vien detto un sistema di numerazione posizionale.

Regola generale

Un numero N sarà rappresentato in base *b* da

$$N = a_0 \cdot b^0 + a_1 \cdot b^1 + a_2 \cdot b^2 + \dots a_n \cdot b^n$$

ove

$$0 \leqslant a_i \leqslant b$$

Casi particolari

– Se b = 10,

$$N = a_0 \cdot 1 + a_1 \cdot 10 + a_2 \cdot 10^2 + \dots a_n \cdot 10^n$$

ove

$$0 \leqslant a_i \leqslant 10$$

a_0 è la cifra delle unità, a_1 quella delle decine, a_2 quella delle centinaia, e così via.
Le a_1 corrispondono ad un carattere decimale.

– Se b = 2,

$$N = a_0 \cdot 1 + a_1 \cdot 2 + a_2 \cdot 2^2 + \dots a_n \cdot 2^n$$

ove

$$0 \leqslant a_i \leqslant 2$$

Le a_i corrispondono ad un carattere binario.

Esempi

1) Il numero decimale 1789 è interpretato come il valore
$9 \cdot 1 + 8 \cdot 10 + 7 \cdot 10^2 + 1 \cdot 10^3 = 9 + 80 + 700 + 1000$

2) Il numero binario 1101 è interpretato come
$1 \cdot 1 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = 1 + 0 + 4 + 8 = 13_{10}$

Questo numero corrisponde nel sistema decimale al numero intero 13.

OSSERVAZIONI

Il considerare il carattere posto all'estrema destra come la cifra dotata del peso minore è una scelta arbitraria, e può sembrare innaturale, dal momento che scriviamo da sinistra a destra. Ma non bisogna dimenticare che il sistema di numerazione per posizione ci è giunto dai matematici indiani, per la mediazione di quelli arabi, e indiani ed arabi scrivono da destra a sinistra.

Nel sistema di numerazione binario, la cifra binaria prende il nome di *bit* (che è una contrazione delle parole inglesi *binary digit*: cifra binaria).

PROPRIETÀ

Da quanto precede segue che una parola binaria di n bits permette di rappresentare i numeri interi che soddisfano la seguente condizione:

$$0 \leqslant N \leqslant 2^{n-1}$$

Esempio

Una parola di otto bits permette di rappresentare gl'interi compresi fra 0 e $2^8 - 1 = 255$.

Numero di bits necessario per rappresentare un numero intero

Inversamente, per rappresentare in binario un numero N, occorrerà un numero di $n = [log_2 N]$ bits, ove il simbolo [ ] indica l'intero immediatamente superiore al valore calcolato.

Se non si hanno a disposizione le tavole dei logaritmi, il numero n in pratica si può determinare individuando le potenze di 2 fra le quali N è compreso:

$$2^{n-1} \leqslant N \leqslant 2^n$$

In questo caso, si sa che per rappresentare N occorrono n bits.

Esempio

Se $N = 746$, avremo:

$$n = [\log_2 N] = 10$$

Infatti,

$$2^9 = 512 < 746 < 2^{10} = 1024$$

Conversione dal sistema binario al sistema decimale

La conversione di un numero binario in un numero decimale si ottiene facendo riferimento alle regole di numerazione.

L'algoritmo di conversione è riassumibile come segue:

1. Posizionarsi sulla destra del numero.
2. Inizializzare *N* a 0 ed *n* a 0.
3. Leggere il carattere che segue procedendo verso sinistra
4. *Se* questo carattere è uno spazio bianco, *allora* fine della procedura.
5. Se questo carattere è zero, *allora* fare $n + 1$ e proseguire dal punto 3, *sennò* calcolare $N + 2^n$, e proseguire dal punto 3.

Esempio

1 0 1 1 1 0

rappresenta, nel sistema decimale, il numero

$$N = 2 + 4 + 8 + 32 = 46$$

Si tenga presente che:
– i numeri pari terminano con un bit 0;
– i numeri dispari terminano con un bit 1.

Conversione dal sistema decimale al sistema binario

Consideriamo la divisione per 2 di un numero intero *N*. Avremo:

$$N = 2 \cdot q_0 + r_0 \qquad 0 \leqslant r_0 \leqslant 1$$

Se $q_0 > 2$, si ha:

$$q_0 = 2q_1 + r_1 \qquad 0 \leqslant r_1 \leqslant 1$$

cioè

$$N = 2^2q_1 + 2r_1 + r_0$$

Se $q_1 > 2$, si ha:

$$q_1 = 2q_2 + r_2 \qquad 0 \leqslant r_2 \leqslant 1$$

cioè

$$N = 2^3q_2 + 2^2 \cdot r_2 + 2 \cdot r_1 + r_0$$

Continuando a dividere per 2 fin quando è possibile i quozienti via via ricavati, otteniamo:

$$N = 2^i q_i + 2^{i-1} r_{i-1} + \dots + 2^3 r_3 + 2^2 r_2 + 2 \cdot r_1 + r_0$$

ove

$$0 \leqslant q_i \leqslant 1$$

I resti delle divisioni successive per 2 dei quozienti rappresentano per definizione le cifre binarie relative al numero *N*.

Esempio

Si debba convertire $N = 59$. Le divisioni successive saranno le seguenti:

59	2				
1	29	2			
	1	14	2		
		0	7	1	
			1	3	2
				1	1

cioè, in binario si avrà 111011. Si tenga presente che i resti vengono riletti dal basso verso l'alto.

Le rappresentazioni ottale ed esadecimale

Nella pratica i numeri binari sono difficili da maneggiare, a causa della loro lunghezza e della loro monotonia. Non sono altro che una sequenza di 0 e 1!

Ma è possibile condensarli in rappresentazioni più compatte.

La rappresentazione ottale

Fa capo al sistema di numerazione in base 8, il cui alfabeto è il seguente:

$$A = \{0, 1, 2, 3, 4, 5, 6, 7\}$$

Il passaggio dal binario all'ottale è immediato, dal momento che si attua considerando i bits che costituiscono il numero binario a tre a tre, in sequenza:

000	0	100	4
001	1	101	5
010	2	110	6
011	3	111	7

Ad esempio, il numero binario 101110 in ottale sarà 56, perchè 101 = 5 e 110 = 6.

Altrettanto immediata è la conversione inversa: ad ogni cifra ottale si sostituisce l'equivalente in binario.

La rappresentazione esadecimale

Fa capo al sistema di numerazione in base 16, il cui alfabeto è il seguente:

0, 1, 2 ... 9, A, B, C, D, E, F

Decimale	Esadecimale	Binario
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Il passaggio dal binario all'esadecimale si attua prendendo questa volta quattro bits successivi, e sostituendoli con una cifra esadecimale. Ad esempio, il numero binario 10111001 diventa in esadecimale B9, perchè 1011 = B e 1001 = 9.

Analogamente, per la conversione inversa, ciascuna cifra esadecimale corrisponde ad una sequenza di quattro bits.

Decimale codificato in binario (BCD)

Alcuni calcolatori e calcolatrici, pur lavorando internamente su parole del linguaggio binario, per rappresentare i numeri decimali ricorrono ad un *codice* che associa a ciascuna cifra decimale la sua rappresentazione in binario, in questo modo:

0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001

Qui per rappresentare i numeri binari *non si fa intervenire* il principio della numerazione binaria oltre la cifra 9: in questo modo ciascuna cifra di un numero è espressa dal suo codice.

Esempi

1) 78 è espresso da 01111000
2) 99 è espresso da 10011001

Ciò vuol dire che non tutte le configurazioni binarie di una parola sono decifrabili in BCD: sono infatti indecifrabili tutte le configurazioni a quattro bits oltre 1001. Le sequenze di quattro bits che seguono sono tutte non consecutive: 1010, 1011, 1100, 1101, 1110, 1111.

Addizione e moltiplicazione binarie

Addizione

Le regole dell'addizione binaria sono analoghe a quelle dell'addizione decimale. La tabella dell'addizione in binario prevede soltanto quattro combinazioni di cifre:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 10$

Solo nel quarto caso si ha riporto: la cifra di risultato è 0, mentre 1 è il riporto.

Esempio

```
  11
 1011       11
+1101      +13
-----      ---
11000       24
```

OSSERVAZIONI

Quando la dimensione delle parole è n, il numero massimo che si può avere per numeri rigorosamente positivi è $2^n - 1$. Se si sommano due numeri la cui somma è maggiore di $2^n - 1$, si ha un superamento della capacità aritmetica.

Moltiplicazione

La tabella della moltiplicazione in binario è la seguente:

$0 \times 0 = 0$
$0 \times 1 = 0$
$1 \times 0 = 0$
$1 \times 1 = 1$

Solo nell'ultimo caso si ha un risultato non nullo.

Algoritmo di moltiplicazione

1. Posizionarsi sulla destra del moltiplicatore.
2. Inizializzare N a 0.
3. Inizializzare il risultato P a 0.
4. Leggere il carattere che segue procedendo verso la sinistra del moltiplicatore.
5. *Se* il carattere è uno spazio bianco, *allora* fine procedimento.
6. *Se* il carattere è 0, fare $N + 1$ e proseguire dal punto 4.
7. *Sennò,* sommare a P il moltiplicando seguito da n zeri, fare $N + 1$ e proseguire dal punto 4.

Esempio

```
     1011
x     101
---------
     1011
   1011
---------
   110111
```

Attenzione: la moltiplicazione di due parole di n bits necessita di una parola di $2n$ bits per il risultato.

Codifica dei numeri negativi

La rappresentazione dei numeri interi negativi implica la codifica del *segno*. A tal fine è necessario un bit: ad esempio, si può prendere il bit più a sinistra di una parola di n bits e, se è 0, dargli il significato +, se è 1, il significato −.

Allora i rimanenti $n - 1$ bits servono a rappresentare il valore assoluto.

Esempi

1) La parola 00000101 rappresenta +5.

2) La parola 10000101 rappresenta −5.

Questo metodo di codifica presenta diversi svantaggi:
- per eseguire un'operazione su degl'interi, positivi e negativi, bisogna verificare sistematicamente il bit di segno;
- occorre definire un'operazione di sottrazione fra valori assoluti;
- infine, la configurazione 10000000 è interpretata come uno zero negativo.

Quindi, con questo metodo, due diversi codici esprimono entrambi lo zero, e questo è un grosso inconveniente.

La rappresentazione in complemento a 1

Convenzionalmente, per rappresentare un numero negativo, si assume il numero opposto ad un altro numero, il che si ottiene prendendo il complemento a 1 di ciascun bit.

Si tenga presente che il complemento a 1 di 0 è 1, mentre il complemento a 1 di 1 è 0.

Esempio

+6	0110
−6	1001

Anche questa convenzione, per rappresentare il segno, si serve del bit più a sinistra.

L'operazione di negazione è semplice, e identica per tutti i bits. L'addizione a due a due dei bits di un numero e del suo opposto dà la configurazione 1111, che è il complemento a 1 di 0000.

Anche in questa rappresentazione si ha l'inconveniente del doppio zero (+0 e −0).

La rappresentazione in complemento a 2

Consideriamo la sottrazione fra due cifre decimali $9 - 3 = 6$. Se consideriamo il complemento a 10 di 3, cioè 7, ed eseguiamo $9 + 7$, otteniamo 16, cioè la cifra 6 associata al riporto di 1. Supponiamo d'ignorare il riporto: la sottrazione di 3 da 9 equivarrà alla somma senza riporto di 9 più il complemento a 10 di 3, cioè 7.

Questo è un principio generale che, nel caso del sistema binario, prende il nome di complemento a 2.

Per avere il complemento a 2 di un numero binario, basta calcolare il complemento a 1 ed aggiungergli 1.

Esempio

Si abbia il numero binario 00001011. Il suo complemento a 1, cioè 11110100, si ottiene sostituendo a ciascun bit il suo opposto.

Il complemento a 2, cioè 11110101, si ottiene aggiungendo + 1 al complemento a 1.

Allora la somma di un numero e del suo opposto sarà:

$$\begin{array}{r} 00001011 \\ 11110101 \\ \hline 100000000 \end{array}$$

Vediamo quindi che questa volta il numero che si ottiene su otto bits è 00000000: infatti il riporto si è propagato fino al nono bit, che non fa parte della parola di otto bits.

La rappresentazione che si ottiene è di questo tipo:

$$a + (-a) = 0$$

In questo modo di rappresentazione il numero binario 10000000 è, per definizione, uguale a −128; infatti è un numero negativo, il cui valore assoluto è 10000000, cioè $2^7 = 128$.

Quindi la rappresentazione in complemento a 2 prevede un solo zero.

Generalizzazione

La rappresentazione in complemento a 2 è quella usata attualmente su tutti i calcolatori.

Per una parola di *n* bits, i numeri rappresentabili in complemento a 2 sono tali per cui

$$-2^{n-1} \leqslant N \leqslant 2^{n-1} - 1$$

Esempio

Con un calcolatore che lavori su parole di sedici bits, si ha:

$$-2^{15} \leqslant N \leqslant 2^{15} - 1$$

cioè

$$-32768 \leqslant N \leqslant 32767$$

La rappresentazione dei numeri reali

Il trattamento dei numeri frazionari richiede che si definisca un altro metodo di rappresentazione dei numeri. Si hanno diverse soluzioni.

La rappresentazione a virgola fissa

Un primo modo può essere quello di considerare una parte intera ed una parte decimale, separate da una virgola la cui posizione è fissata a priori.

In una parola di 8 bits, possiamo stabilire la posizione della virgola al centro della parola, avendo così quattro bits per la parte intera e quattro per la parte decimale.

$$N = a_3 . 2^3 + a_2 . 2^2 + a_1 . 2 + a_0 + a_{-1} . 2^{-1} + a_{-2} . 2^{-2} + a_{-3} \times . 2^{-3} + a_{-4} . 2^{-4}$$

In questo modo ci è consentito rappresentare numeri positivi frazionari compresi fra 0000.0000 ed 1111.1111, cioè fra 0.0 e 15.9375, che è un intervallo di variazione molto ridotto.

Se poi prendiamo in considerazione sia i numeri positivi che i numeri negativi, l'intervallo di rappresentazione sarà ancora più ristretto: da —8.9375 a +7.9375.

Si vede quindi che la rappresentazione a virgola fissa è molto limitata, in ragione del numero di bits su cui si può lavorare.

La rappresentazione a virgola mobile

Un'altra soluzione consiste nell'applicare al numero frazionario una rappresentazione esponenziale (logaritmica).

Ad esempio, il numero

1984.128

può venir rappresentato in diverse forme equivalenti:

$19.84128 \cdot 10^2$
$0.1984128 \cdot 10^4$
$1984128 \cdot 10^{-3}$

Variando il valore della potenza, otteniamo lo spostamento della virgola: di qui il nome di virgola mobile.

In binario, questo tipo di rappresentazione si traduce in un numero binario moltiplicato per una potenza di 2. Ad esempio,

$$
\begin{aligned}
& 11011.1101 = \\
= \; & 110111101 \cdot 2^{-4} = \\
= \; & 0.110111101 \cdot 2^{5}
\end{aligned}
$$

La rappresentazione normalizzata

La rappresentazione normalizzata è caratterizzata dal fatto che la posizione della virgola, o meglio del punto, è tale per cui la prima cifra significativa si trova alla sua destra.

Esempio

$0.110111101 \cdot 2^5$ è una rappresentazione normalizzata.

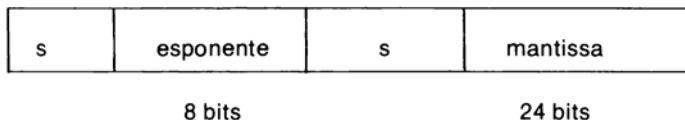
Diamo alcune definizioni.

a) Viene detto *mantissa* l'insieme delle cifre significative (bits) in rappresentazione normalizzata.

b) L'*esponente*, ovvero la caratteristica, è la potenza di 2 che interviene nella rappresentazione normalizzata.

c) Un numero mobile è caratterizzato dalla presenza di un numero binario (positivo o negativo) in qualità di mantissa, e di un numero binario, positivo o negativo, che costituisce l'esponente.

Esempio

Possiamo definire un numero a virgola mobile mediante la struttura seguente:

Rappresentazione dei caratteri

All'interno di un sistema, la rappresentazione dei caratteri avviene per mezzo di un codice.

Per rappresentare l'insieme delle lettere dell'alfabeto e l'insieme delle cifre, cioè complessivamente 36 caratteri, sono necessari 8 bits, perchè

$$32 < 36 < 64 = 2^6$$

Se poi si aggiungono le minuscole ed i caratteri speciali, ne serviranno 7.

Storicamente, i primi ad essere usati sono stati i codici a 6 bits. Successivamente si è imposto il codice a 7 bits, con un bit di parità (il codice ASCII). Alcuni costruttori utilizzano poi un codice ad 8 bits (il codice EBCDIC).

Attualmente, i codici praticamente usati sono i codici ad 8 bits (ASCII ed EBCDIC).

Ed ecco la tabella dei codici ASCII:

Caratteri di controllo	ASCII (esadecimale)	Caratteri speciali e cifre	ASCII (esadecimale)	Caratteri maiuscoli	ASCII (esadecimale)	Caratteri minuscoli	ASCII (esadecimale)
NUL	00	SP (spazio)	20	@	40	/	60
SOH	01	!	21	A	41	a	61
STX	02	"	22	B	42	b	62
ETX	03	#	23	C	43	c	63
EOT	04	$	24	D	44	d	64
ENQ	05	%	25	E	45	e	65
ACK	06	&	26	F	46	f	66
BEL	07	/	27	G	47	g	67
BS	08	(	28	H	48	h	68
HT	09	)	29	I	49	i	69
LF	0A	*	2A	J	4A	j	6A
VT	0B	+	2B	K	4B	k	6B
FF	0C	,	2C	L	4C	l	6C
CR	0D	–	2D	M	4D	m	6D
SO	0E	.	2E	N	4E	n	6E
SI	0F	/	2F		4F	o	6F
DLE	10	0	30	P	50	p	70
DCA (X-ON)	11	1	31	Q	51	q	71
DC2 (TAPE)	12	2	32	R	52	r	72
DC3 (X-OFF)	13	3	33	S	53	s	73
DC4 (TAPE)	14	4	34	T	54	t	74
NAK	15	5	35	U	55	u	75
SYN	16	6	36	V	56	v	76
ETB	17	7	37	W	57	w	77
CAN	18	8	38	X	58	x	78
EM	19	9	39	Y	59	y	79
SUB	1A	:	3A	Z	5A	z	7A
ESC	1B	;	3B	[	5B	{	7B
FS	1C	<	3C	/	5C	I	7C
GS	1D	=	3D	]	5D	}(ALT MODE)	7D
RS	1E	>	3E	Δ(↑)	5E		7E
US	1F	?	3F	– (←)	5F	DEL	7F

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 338 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 676 950 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125

POTENZE DI 16 (IN BASE 10)

16^n	n	16^{-n}	
1	0	0.10000 00000 00000 00000	× 10
16	1	0.62500 00000 00000 00000	× 10^{-1}
256	2	0.39062 50000 00000 00000	× 10^{-2}
4 096	3	0.24414 06250 00000 00000	× 10^{-3}
65 536	4	0.15258 78906 25000 00000	× 10^{-4}
1 048 576	5	0.95367 43164 06250 00000	× 10^{-6}
16 777 216	6	0.59604 64477 53906 25000	× 10^{-7}
268 435 456	7	0.37252 90298 46191 40627	× 10^{-8}
4 294 967 296	8	0.23283 06436 53869 62891	× 10^{-9}
68 719 476 736	9	0.14551 91522 83668 51807	× 10^{-10}
1 099 511 627 776	10	0.90949 47017 72928 23792	× 10^{-12}
17 592 186 044 416	11	0.56843 41886 08080 14870	× 10^{-13}
281 474 976 710 656	12	0.35527 13678 80050 09294	× 10^{-14}
4 503 599 627 370 496	13	0.22204 46049 25031 30808	× 10^{-15}
72 057 594 037 927 936	14	0.13877 78780 78144 56755	× 10^{-16}
1 152 921 504 606 846 976	15	0.86736 17379 88403 54721	× 10^{-18}

POTENZE DI 10 (IN BASE 16)

			10^n	n			10^{-n}		
			1	0	1.0000	0000	0000	0000	
			A	1	0.1999	9999	9999	999A	
			64	2	0.28F5	C28F	5C28	F5C3	× 16^{-1}
			3E8	3	0.4189	374B	C6A7	EF9E	× 16^{-2}
			2710	4	0.68DB	8BAC	710C	B296	× 16^{-3}
		1	86A0	5	0.A7C5	AC47	1B47	8423	× 16^{-4}
		F	4240	6	0.10C6	F7A0	B5ED	8D37	× 16^{-4}
		98	9680	7	0.1AD7	F29A	BCAF	4858	× 16^{-5}
		5F5	E100	8	0.2AF3	1DC4	6118	73BF	× 16^{-6}
		3B9A	CA00	9	0.44B8	2FA0	9B5A	52CC	× 16^{-7}
	2	540B	E400	10	0.6DF3	7F67	5EF6	EADF	× 16^{-8}
	17	4876	E800	11	0.AFEB	FF0B	CB24	AAFF	× 16^{-9}
	E8	D4A5	1000	12	0.1197	9981	2DEA	1119	× 16^{-9}
	918	4E72	A000	13	0.1C25	C268	4976	81C2	× 16^{-10}
	5AF3	107A	4000	14	0.2D09	370D	4257	3604	× 16^{-11}
3	8D7E	A4C6	8000	15	0.480E	BE7B	9D58	566D	× 16^{-12}
23	8652	6FC1	0000	16	0.734A	CA5F	6226	F0AE	× 16^{-13}
163	4578	5D8A	0000	17	0.B877	AA32	36A4	B449	× 16^{-14}
DE0	B6B3	A764	0000	18	0.1272	5DD1	D243	ABA1	× 16^{-14}
8AC7	2304	89E8	0000	19	0.1D83	C94F	B6D2	AC35	× 16^{-15}

APPENDICE 2

SINTASSI DEL LINGUAGGIO PASCAL

Identificatore
Lettera
Lettera
Cifra
Programma
PROGRAM
Identificatore
(
Identificatore
)
;
Blocco
.
,
Blocco
LABEL
Numero intero senza segno
,
;
CONST
Identificatore
=
Costante
;
TYPE
Identificatore
=
Tipo
;
VAR
Identificatore
:
Tipo
,
;
;
Blocco
;
PROCEDURE
Identificatore
Elenco di parametri
FUNCTION
Identificatore
Elenco di parametri
;
Identificatore di tipo
BEGIN
Istruzione
END
;

Istruzione
Numero intero senza segno
:
Variabile
:=
Espressione
Identificatore di funzione
,
Identificatore di procedura
(
Espressione
)
Identificatore di procedura
BEGIN
Istruzione
END
;
IF
Espressione
THEN
Istruzione
ELSE
Istruzione
CASE
Espressione
OF
Costante
:
Istruzione
END
,
;
WHILE
Espressione
DO
Istruzione
REPEAT
Istruzione
UNTIL
Espressione
;
DOWNTO
FOR
Identificatore di variabile
:=
Espressione
TO
Espressione
DO
Istruzione
,
WITH
Variabile
DO
Istruzione
GOTO
Numero intero senza segno

Tipo semplice

Tipo

Variabile
Identificatore di variabile
Identificatore di campo
[
Espressione
]
,
.
Identificatore di campo
↑
Elenco di parametri
(
;
Identificatore
:
Identificatore di tipo
)
FUNCTION
,
VAR
,
PROCEDURE
Identificatore
Elenco di campi
;
,
Identificatore
:
Tipo
CASE
Identificatore
:
Identificatore di tipo
OF
;
,
Costante
:
(
Elenco di campi
)

Fattore

Costante senza segno

Variabile

Identificatore di funzione

(

Espressione

)

,

(

Espressione

)

NOT

Fattore

[

]

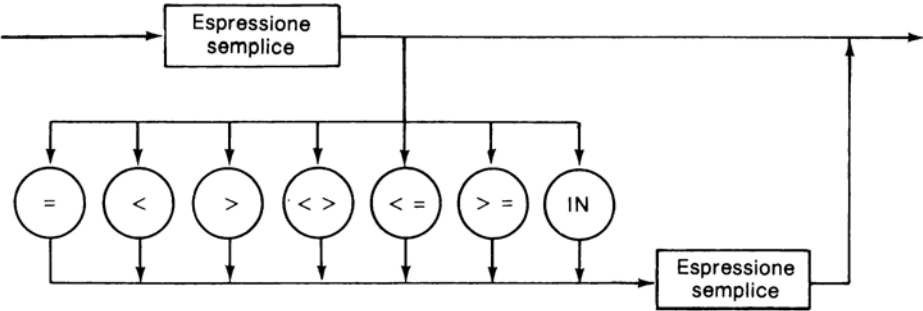
Espressione

..

Espressione

,

Espressione semplice

Espressione

APPENDICE 3

LE PAROLE RISERVATE E GL'IDENTIFICATORI STANDARD IN PASCAL

1 - Le parole riservate

Diamo l'elenco delle parole riservate del linguaggio Pascal standard:

and	mod
array	nil
begin	not
case	of
const	or
div	packed
do	procedure
downto	program
else	record
end	repeat
file	set
for	then
forward	to
function	type
goto	until
if	var
in	while
label	with

2 - Gl'identificatori standard

Sono i seguenti:

a) *Costanti*

alfaleng	numero di caratteri per parola
maxint	massimo intero disponibile sul calcolatore
false	falso
true	vero

b) *Flussi d'ingresso/uscita*

input	flusso d'ingresso standard
output	flusso di uscita standard

c) *Tipi*

integer	intero
real	reale
boolean	booleano
char	tutti i caratteri disponibili
text	flusso testo
alfa	stringa di alfaleng caratteri
string	stringa di caratteri (U.C.S.D.)

d) *Procedure*

Le procedure seguite dall'asterisco non sono completamente standardizzate.

dispose(p)*	rilascio dinamico di un puntatore
get(f)	trasferimento di una componente del flusso f nella variabile buffer f↑
mark*	marcatura del puntatore di pila
new(p)	allocazione dinamica (puntatore)
pack	compattamento di vettori
page(f)	salto di una pagina in un flusso
put(f)	scrittura del valore della variabile buffer f↑ nel flusso f
read	lettura senza andata a capo
readln	lettura con andata a capo
release(p)*	rilascio (puntatore di pila)
reset(f)	apertura di un flusso in lettura
rewrite(f)	apertura di un flusso in scrittura
unpack	disimpaccamento di vettori
write	scrittura
writeln	scrittura con andata a capo

e) *Funzioni*

abs(x)	valore assoluto di x
arctan(x)	arctg x. In alcuni sistemi si usa atan
chr(x)	x-esimo carattere
cos(x)	cos x
eof(f)	fine flusso
eoln(f)	fine linea
exp(x)	e^x
ln(x)	log neperiano di x
odd(x)	parità di x: vero se x è dispari
ord(x)	ordinale di x
pred(x)	predecessore di x
round(x)	arrotondamento di x
sin(x)	sen x
sqr(x)	quadrato di x, cioè x^2
sqrt(x)	radice quadrata di x, cioè $\sqrt{x}$
succ(x)	successore di x
trunc(x)	troncamento di x

APPENDICE 4

CODICI DEGLI ERRORI STANDARD

I codici seguiti dall'asterisco sono tipici del sistema U.C.S.D. (Apple).

1: errore su tipo semplice
2: è richiesto un identificatore
3: è richiesto 'program'
4: è richiesto ')'
5: è richiesto ':'
6: simbolo non permesso
7: errore nell'elenco parametri
8: è richiesto 'of'
9: è richiesto '('
10: errore su tipo
11: è richiesto '['
12: è richiesto ']'
13: è richiesto 'end'
14: è richiesto ';'
15: è richiesto un intero
16: è richiesto '='
17: è richiesto 'begin'
18: errore nella sezione dichiarazione
19: errore nell'elenco campi
20: è richiesto '.'
21: è richiesto ' '
22:* è richiesto 'interface'
23:* è richiesto 'implementation'
24:* è richiesto 'unit'
50: errore in una costante
51: è richiesto ':='
52: è richiesto 'then'

53: è richiesto 'until'
54: è richiesto 'do'
55: è richiesto 'to'/'downto'
56: è richiesto 'if'
57: è richiesto 'file'
58: errore di fattore
59: errore di variabile
101: identificatore dichiarato due volte
102: limite inferiore superiore al limite superiore
103: identificatore non della classe corretta
104: identificatore non dichiarato
105: non è permesso il segno
106: è richiesto un numero
107: tipi sottocampo incompatibili
108: flusso non permesso qui
109: il tipo non dev'essere reale
110: il campo discriminatore dev'essere scalare o sottocampo
111: non compatibile con il campo discriminatore
112: l'indice non può essere reale
113: l'indice dev'essere scalare o sottocampo
114: il tipo base non dev'essere reale
115: il tipo base dev'essere scalare o sottocampo
116: il tipo del parametro di una procedura standard è errato
117: riferimento 'forward' non soddisfatto
118: riferimento 'forward' per identificatore tipo in dichiarazione variabile
119: dichiarato 'forward'; non è concesso ripetere elenco parametri;
120: il tipo del risultato di una funzione dev'essere scalare, sottocampo o puntatore
121: non è permesso un valore flusso come parametro
122: funzione dichiarata 'forward'; non è permessa la ripetizione del tipo del risultato
123: è stato omesso il tipo del risultato nella dichiarazione di funzione
124: formato F solo per reali
125: è errato il tipo di un parametro di una funzione standard
126: il numero dei parametri non si accorda con la dichiarazione
127: sostituzione non legale di parametri
128: non c'è accordo tra la dichiarazione ed il tipo del risultato di un parametro funzione
129: conflitto tra i tipi degli operandi
130: l'espressione non è di tipo set
131: è concessa solo la verifica di disuguaglianza
132: non è concessa inclusione stretta
133: non è concesso confronto di flussi
134: il tipo degli operandi non è tra quelli consentiti

135: il tipo degli operandi dev'essere booleano
136: il tipo degli elementi di un insieme dev'essere scalare o sottocampo
137: i tipi degli elementi di un insieme non sono compatibili
138: la variabile non è di tipo vettore
139: il tipo dell'indice non è compatibile con la dichiarazione
140: la variabile non è di tipo record
141: la variabile dev'essere di tipo flusso o puntatore
142: sostituzione non legale di parametri
143: la variazione di controllo della ripetizione è di tipo non consentito
144: espressione di tipo non consentito
145: conflitto di tipi
146: non è concessa l'assegnazione a flussi
147: il tipo dell'etichetta è incompatibile con l'espressione di selezione
148: i limiti di un sottocampo devono essere scalari
149: l'indice non può essere di tipo intero
150: non è concessa l'assegnazione a funzioni standard
151: non è concessa l'assegnazione a funzioni formali
152: non c'è un campo come questo nel record
153: errore di tipo in lettura
154: il parametro effettivo dev'essere una variabile
155: la variabile di controllo non dev'essere dichiarata in un livello intermedio
156: definizione multipla di etichetta in una selezione
157: troppi casi in un'istruzione di selezione
158: omessa la dichiarazione della variante corrispondente
159: non sono permessi reali o stringhe come campi selezionatori
160: la dichiarazione precedente non era 'forward'
161: ancora dichiarata 'forward'
162: la dimensione del parametro dev'essere costante
163: variante omessa nella dichiarazione
164: non è permessa la sostituzione di procedure/funzioni standard
165: etichetta con definizioni multiple
166: etichetta con dichiarazioni multiple
167: etichetta non dichiarata
168: etichetta non definita
169: errore nell'insieme base
170: è richiesto un parametro valore
171: flusso standard ri-dichiarato
172: flusso esterno non dichiarato
173: è richiesta una procedura o funzione FORTRAN
174: è richiesta una procedura o funzione Pascal
175: omesso il flusso 'input' nell'intestazione del programma
176: omesso il flusso 'output' nell'intestazione del programma
177: qui non sono concesse assegnazioni a identificatori di funzione

178: variante di record con definizioni multiple
179: X-opt di procedure/funzioni effettive non in accordo con dichiarazione formale
180: la variabile di controllo non dev'essere formale
181: la parte costante indirizzo fuori dei limiti
201: errore in costante reale: è richiesta cifra
202: una costante stringa non deve superare una linea
203: costante intera oltre i limiti
204: 8 o 9 in numero ottale
205: ci dev'essere almeno una stringa
206: parte intera di una costante reale oltre i limiti
250: troppi annidamenti d'identificatori
251: troppi annidamenti di procedure e/o funzioni
252: troppi riferimenti 'forward' di ingressi di procedure
253: procedura troppo lunga
254: costanti troppo lunghe in questa procedura
255: troppi errori in questa linea sorgente
256: troppi riferimenti esterni
257: troppi dati esterni
258: troppi flussi locali
259: espressione troppo complicata
260: troppe etichette d'uscita
300: divisione per zero
301: non è previsto un caso per questo valore
302: espressione dell'indice oltre i limiti
303: il livello che va assegnato è oltre i limiti
304: elemento dell'espressione non nei limiti
350:* segmento dato non consentito
351:* segmento dichiarato due volte
352:* segmento codice non consentito
353:* unità intrinseca non chiamata
398: restrizione implementativa
399: non sono implementate dimensioni variabili per vettori
400:* carattere non lecito nel testo
401:* fine inaspettata dell'ingresso
402:* errore nella scrittura del flusso codice per spazio insufficiente
403:* errore nella lettura di un flusso
404:* errore nella scrittura di un flusso per spazio insufficiente
405:* chiamata non consentita in una procedura distinta
406:* inclusione di flussi non lecita
407:* troppe biblioteche

L. 39.000

Cod. 516 A ISBN 88-7056-118-6

Pierre Le Beux, dopo aver insegnato a un grandissimo numero di persone il Basic con il libro "Introduzione al Basic", introduce ora allo studio del Pascal, il linguaggio destinato a spodestare negli anni 80 il FORTRAN, i vari derivati dell'ALGOL, il PL/I ecc.
Il volume, mantenendo quelle caratteristiche del libro precedente che ne hanno decretato il successo e che ne fanno un vero e proprio corso (chiarezza espositiva, trattazione incentrata su numerosissimi esempi che verificano costantemente l'apprendimento del lettore), insegna a conoscere, capire ed usare tutte le particolarità e i vantaggi di questo linguaggio (nel corso della trattazione vengono ampiamente utilizzate le tecniche di programmazione strutturata, come pure tecniche particolari, quali il trattamento dei file, utilizzazione della recursività e trattamento grafico), con riferimento particolare al Pascal sviluppato presso l'Università di San Diego in California (UCSD), disponibile su diversi sistemi a microprocessore.
È un'opera completa che si presta ad essere letta dal piancipiante come da chi ha già familiarità con altri linguaggi.

Pierre Le Beux

Introduzione al PASCAL

61